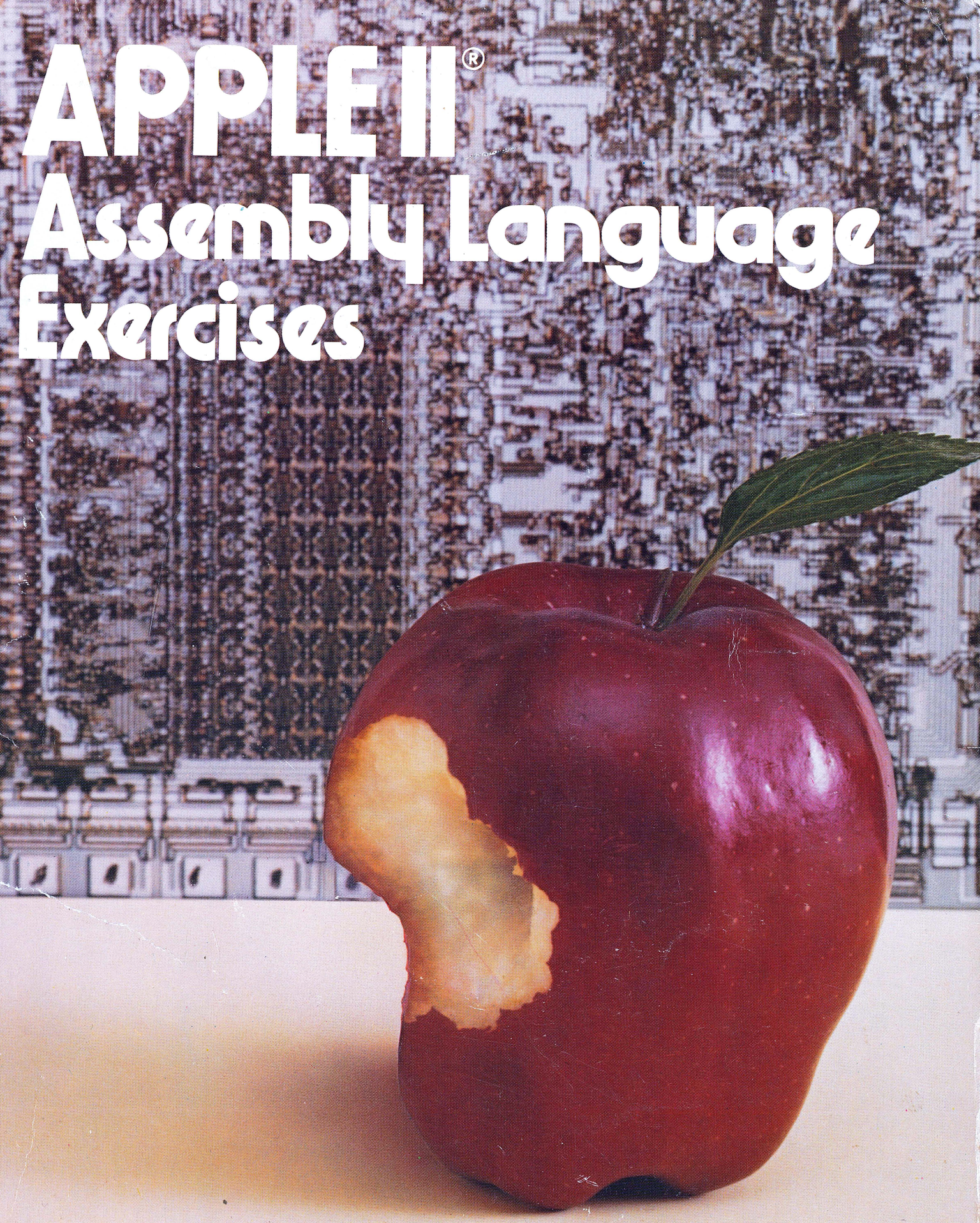
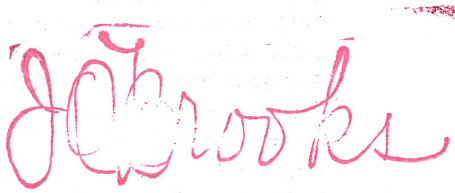
APPLE II®
Assembly Language Exercises



Leo J. Scanlon

Apple II® Assembly Language Exercises



MORE THAN A MILLION PEOPLE HAVE LEARNED TO PROGRAM, USE, AND ENJOY MICROCOMPUTERS WITH WILEY PAPERBACK GUIDES. LOOK FOR THEM ALL AT YOUR FAVORITE BOOKSHOP OR COMPUTER STORE:

ANS COBOL, 2nd ed., Ashley
Apple® BASIC: Data File Programming, Finkel & Brown
Apple II® Assembly Language Exercises, Scanlon
8080/Z80 Assembly Language, Miller
6502 Assembly Language Programming, Fernandez, Tabler, & Ashley
ATARI® BASIC, Albrecht, Finkel & Brown
ATARI® Sound and Graphics, Moore, Lower, & Albrecht
Background Math for a Computer World, 2nd ed., Ashley
BASIC, 2nd ed., Albrecht, Finkel, & Brown
BASIC for Home Computers, Albrecht, Finkel, & Brown
BASIC for the Apple II®, Brown, Finkel, & Albrecht
BASIC Programmer's Guide to Pascal, Borgerson
***BASIC Subroutines for the PET®,** Adamis
Byteing Deeper into Your Timex Sinclair 1000, Harrison
Color Computer ™ Applications, Grillo & Robertson
***Complete BASIC Dictionary,** Adamis
Data File Programming in BASIC, Finkel & Brown
FAST BASIC: Beyond TRS-80™ BASIC, Gratzer
Flowcharting, Stern
FORTRAN IV, 2nd ed., Friedmann, Greenberg, & Hoffberg
Fundamentals of Microcomputer Programming including Pascal, McGlynn
Genie in the Computer: Programming Graphics on the TRS-80™, Kohl, Karp, & Singer
Golden Delicious Games for the Apple® Computer, Franklin, Finkel, & Koltnow
How to Buy the Right Small Business Computer System, Smolin
Introduction to 8080/8085 Assembly Language Programming, Fernandez & Ashley
Introduction to Computer Music, Bateman
Introduction to Data Processing, 2nd ed., Harris
Job Control Language, Ashley & Fernandez
More Subroutine Sandwich, Grillo & Robertson
More TRS-80™ BASIC, Inman, Zamora, & Albrecht
Personal Computing, 2nd ed., McGlynn
Problem-Solving on the TRS-80 ™ Pocket Computer, Inman & Conlan
Structured COBOL, Ashley
Subroutine Sandwich, Grillo & Robertson
Successful Software for Small Computers, Beech
The TRS-80™ Model III Users Guide, Finkel & Bove
TRS-80™ BASIC, Albrecht, Inman & Zamora
TRS-80™ Color BASIC, Albrecht
TRS-80™ Means Business, Lewis
Using CP/M®, Fernandez & Ashley
Using Programmable Calculators for Business, Hohenstein
***Visicalc Applications,** Klitzner
Why Do You Need A Personal Computer?, Leventhal & Stafford

*Forthcoming.

Apple® is a registered trademark of Apple Computer, Inc.
ATARI® is a registered trademark of Atari, Inc.
CP/M® is a registered trademark of Digital Research.
PET® is a registered trademark of Commmodore International.
TRS-80™ is a trademark of Tandy Corp.

Apple II® Assembly Language Exercises

LEO J. SCANLON

John Wiley & Sons, Inc.
New York • Chichester • Brisbane • Toronto • Singapore

Apple, Apple II, Apple II Plus, Disk II, and Applesoft are registered trademarks of Apple Computer Inc.

Copyright © 1982, by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by Section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging in Publication Data

Scanlon, Leo J.
Apple II assembly language exercises.

1. Apple II (Computer)—Programming—Problems, exercises, etc. 2. Assembler language (Computer program language)—Problems, exercises, etc. I. Title.
QA76.8.A662S3 001.64'2 82-2822
ISBN 0-471-86598-2 AACR2

Printed in the United States of America

82 83 10 9 8 7 6 5 4 3 2 1

Contents

Preface		vii
Acknowledgments		ix
Chapter 1	**Getting Inside the Apple II**	1
Chapter 2	**Using an Assembler**	22
Chapter 3	**Subtraction and Logical Operations**	38
Chapter 4	**Program Sequencing**	45
Chapter 5	**Debugging Programs**	54
Chapter 6	**Multiplication Operations, with Shift and Rotate**	66
Chapter 7	**Division Operations**	73
Chapter 8	**Subroutines and the Stack**	78
Chapter 9	**Unordered Lists**	90
Chapter 10	**Sorting Unordered Data**	96
Chapter 11	**Code Conversion from Input**	101
Chapter 12	**Code Conversion for Output**	109
Chapter 13	**Timing Events with Software**	115
Chapter 14	**Apple II Sound**	120
Chapter 15	**Let There Be Music!**	124
Chapter 16	**More Music**	130
Chapter 17	**Low-Resolution Graphics**	134
Chapter 18	**Reset and Interrupts**	145
	Some Closing Comments	153
Answers to Exercises		155
Appendix A	**Hexadecimal and Decimal Conversion**	197
Appendix B	**Computer Numbering Systems**	198
Appendix C	**ASCII Character Set for Apple II**	203
Appendix D	**Low-Resolution Graphics Screen**	204
Appendix E	**6502 Instruction Set Table**	205

Preface

This book puts the full power of the Apple II® or Apple II Plus® at your fingertips.* The speed and efficiency of *assembly language* are the keys to this power, power largely unavailable through high-level languages such as BASIC.

Who needs to learn to program in assembly language? *You do,* if:

- A task must be performed in the shortest possible time.
- You wish to control the Apple II's operation directly.
- You want to understand fully the *6502 microprocessor,* the "brain" of the Apple II.
- Your computer must communicate with attached equipment in "real time."
- Your application is not covered by software currently on the market.

This book will help you unlock the power of your Apple II with 18 easy-to-read "hands-on" exercises. Beginning with a discussion of the assembly language characteristics of the Apple II, the book guides you through the 6502 microprocessor's instruction set, with meaningful programs each step of the way. This is followed by exercises in processing lists and data tables, and input/output techniques. Finally, you will learn how to develop *music* programs that play through the Apple's built-in speaker (something virtually impossible with BASIC) and low-resolution *graphics* programs, including animation.

After completing the final exercise, you will have developed a library of useful assembly language programs to use in building programs of your own design—and you will have the expertise to do so.

THE SYSTEM NEEDED WITH THIS BOOK

To perform the exercises in this book, you will need an Apple II or Apple II Plus computer with an attached display screen (either a television set or a monitor) and a mass storage device. You will also need a *6502 assembler* software package. Most assemblers come on floppy disk, but some come on cassette, so your mass storage device will be either a Disk II or a cassette recorder.

*Apple II and Apple II Plus are registered trademarks of Apple Computer, Inc.

This book is written around the 6502 Assembler/Editor included in the *DOS Tool Kit,* manufactured by Apple Computer, Inc, but the descriptions are easily adaptable to any other 6502 assembler you might have.

SUPPLEMENTARY REFERENCE BOOKS

This book covers most details of the 6502 instruction set, but you may like to use one of the popular tutorials as a supplement. Some tutorials are:

- Judi Fernandez and Ruth Ashley, *Introduction to 6502 Assembly Language Programming,* John Wiley & Sons, New York, 1982.
- Leo J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980.
- Lance A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979.
- Marvin L. De Jong, *Programming and Interfacing the 6502, with Experiments,* Howard W. Sams, Indianapolis, IN, 1980.
- Don Inman and Kurt Inman, *Apple Machine Language,* Reston, Reston, VA, 1980.

Of course, you should also have a copy of the *Apple II Reference Manual,* which comes with every Apple II.

Although it is not required, readers who wish to explore the details of the Apple II Monitor are advised to get a copy of the *Apple II Monitor Peeled.* This book is available from its author, William E. Dougherty, 14349 San Jose Street, Los Angeles, CA 91345, or from Apple Computer Inc.

The *Apple II User's Guide* by Lon Poole, Martin McNiff, and Steven Cook (Osborne/McGraw-Hill, Berkeley, CA, 1981) is also an excellent addition to your library.

May you have as much satisfaction developing assembly language programs as I have had writing this book.

Leo J. Scanlon

Acknowledgments

I would like to express thanks to Ms. Dianne Littwin of John Wiley & Sons, for keeping me on the right track, and to Mr. Mark Reardon of Anaheim, California, and Mr. Bill Exberger of Tampa, Florida for their support of this project.

Apple II® Assembly Language Exercises

CHAPTER ONE

Getting Inside the Apple II

OBJECT

To become familiar with the Apple II's "brain"—the 6502 microprocessor—and its internal operating program *(Monitor)*.

INSTRUCTIONS DESCRIBED

This chapter describes the *load* instructions (LDA, LDX, and LDY), the *store* instructions (STA, STX, and STY), and the BRK instruction.

PREPARATION

Read Chapters 1 through 4 of the *Apple II Reference Manual*. Concentrate on the operating commands and software descriptions, without being too concerned about hardware descriptions.

DISCUSSION

When you first turn on the power to the Apple II, you should hear a "beep" from inside the computer and the power light on the keyboard should come on. The screen should also be displaying something at this time. What is being displayed on the screen depends on which version of the Apple you have, but it should include the blinking *cursor* and one other character. The Apple II is multilingual, and the character to the left of the cursor indicates that the computer is waiting for instructions in one of its three primary languages. Specifically:

- If an asterisk (*) is displayed, the computer is awaiting monitor instructions. As you shall soon see, the Monitor (not to be confused with the type of display screen called a "monitor") is the Apple's operating program and is used in assembly language programming.
- If a "greater than" (>) symbol is displayed, the computer is awaiting *Integer BASIC* instructions.
- If a right bracket ( ]) is displayed, the computer is awaiting *Applesoft BASIC* instructions.

Later in this chapter you will put the Apple into the monitor mode, but for now leave it in the present mode, whatever that might be.

The keyboard, screen, power switch, and any peripherals connected to your Apple II are external features of the system. If you plan to do any assembly language programming (as you undoubtedly do, having bought this book), you must also be familiar with the "brain" of the Apple II, its *microprocessor*. The microprocessor, an integrated circuit (or *chip*) on the main circuit board inside the chassis, performs all processing operations within the system. Because of its importance, the characteristics of this chip should be examined in detail.

IMPORTANT

In the next section, and throughout the rest of this book, there are references to *binary* numbers and *hexadecimal* numbers. If you are unfamiliar with these terms, read Appendix B before continuing.

The 6502 Microprocessor

A microcomputer comprises a *microprocessor* (usually referred to as a CPU, or central processing unit), memory, and at least one input device and one output device. In the Apple II microcomputer, the microprocessor is a *6502*, an integrated circuit manufactured by Rockwell International, Synertek, and Commodore. The 6502 is an 8-bit microprocessor, which means it operates on instructions and data 8 bits at a time.

The 6502 executes programs by fetching instructions from memory, one by one, and executing them. Some instructions occupy only a single 8-bit location, or *byte*, in memory. Other instructions include an *operand* (a data value or a memory address) and occupy 2 or 3 bytes in memory. The first byte of every instruction is an operation code, or *op-code*, that tells the 6502 which operation to perform.

The sum total of all instructions executable by the 6502 is referred to as its *instruction set*. There are fifty-six different instructions in the 6502 instruction set; that is, the 6502 can recognize fifty-six different op-codes.

In the early days of computers, programmers were required to enter the actual op-codes into memory, which meant that they had to constantly refer to a table of numbers to do their programming. This procedure was not only tedious and time consuming, but was extremely error prone as well. To increase programmers' effi-

TABLE 1–1. 6502 Instruction Mnemonics

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEO	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift One Bit Right (Memory or Accumulator)
NOP	No operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Register
TYA	Transfer Index Y to Accumulator

Source: Courtesy of Rockwell International.

ciency, *assemblers* were developed that could accept abbreviations instead of numeric op-codes, and translate these abbreviations into the appropriate op-codes. In Chapter 2, when you begin using your assembler, you will use the standard abbreviations the manufacturers have assigned to each of the fifty-six op-codes in the 6502 instruction set. These abbreviations, called *mnemonics* (the first "m" is silent), are summarized in Table 1–1.

Unless you have programmed in assembly language before, the descriptions of the mnemonics in Table 1–1 may appear to be written in some strange language in which terms like *accumulator, carry,* and *index X* abound. Are these secret code words concocted by the manufacturers to keep you in the dark? No, these terms refer to physical components of the microprocessor; components that can be manipulated, using assembly language instructions, to make the microprocessor perform according to your wishes. As you can see, then, *assembly language allows you to control the internal operation of the microprocessor.* This degree of control is unattainable through "high-level" languages, such as *BASIC.*

The 6502's Internal Registers

The internal components of the 6502 that can be manipulated with assembly language instructions are the *registers* shown in Figure 1–1. All these registers are 8 bits wide, except the *program counter,* which is 16 bits wide.

The *accumulator* is a general-purpose register in which all data-processing operations are conducted. Whenever you add, subtract, or otherwise combine two data values, one of these values must be contained in the accumulator. The accumulator will always hold the result of these operations, too. The *X and Y registers* are also general-purpose registers, but have no arithmetic capabilities. X and Y are generally used as "index" registers, for accessing data relative to some base address. However, the 6502 also has instructions to increase or decrease these registers by one, making them useful as counters.

The remaining three registers have dedicated functions; they are not general-purpose registers. The *program counter* (PC) holds the memory address of the next

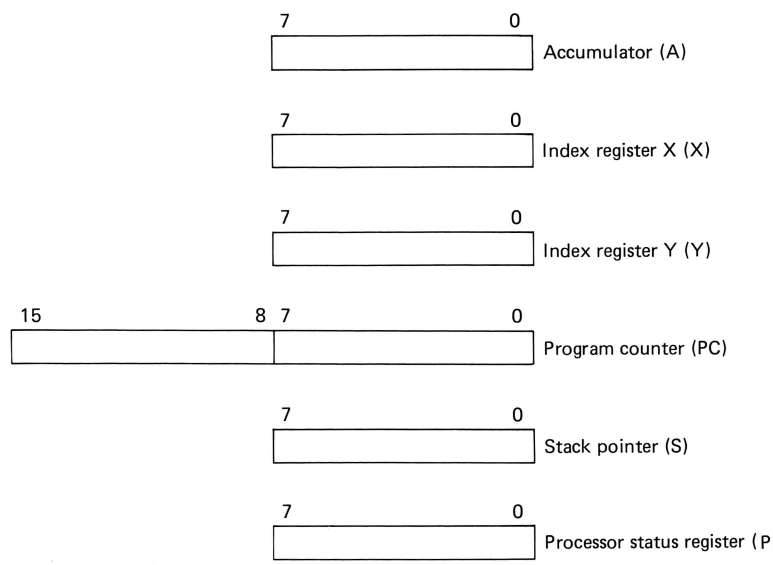


FIGURE 1–1. Programmable Registers in the 6502 Microprocessor

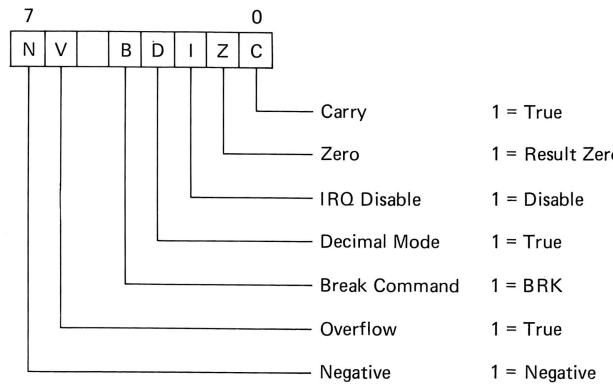


FIGURE 1–2. The Processor Status Register

instruction to be executed. Because it is 16 bits wide, the PC can hold 65,536 different addresses, from 0 to 65,535 decimal (FFFF hexadecimal). Once it has been loaded with the starting address of a program, the microprocessor will update the PC as required. The *stack pointer* will be discussed in Chapter 8. The *processor status register* reflects the current state of seven different conditions within the 6502. This register warrants a more detailed description.

The Processor Status Register

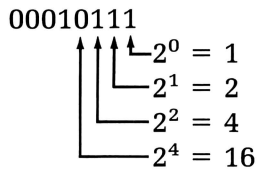
The processor status register's seven usable bits are shown in Figure 1–2. Five of these bits are *status flags;* they provide information on the result of a previously executed instruction (in most cases, the preceding instruction). The two other usable bits are *control bits.*

The carry flag (C) is used to save any carry produced by an add operation, any borrow produced by a subtract operation, or the value of a bit displaced out of an operand during a shift operation. The zero flag (Z) indicates whether the result of a preceding operation is zero. The break command flag (B) indicates whether an interrupt request to the 6502 was caused by a BRK instruction or by an externally generated interrupt; more on these terms later. The overflow flag (V) is applicable only to arithmetic operations on signed numbers. V is set if the addition of two like-signed numbers or the subtraction of two unlike-signed numbers produces a result greater than $+127_{10}$ or less than -128_{10}. The negative flag (N) indicates whether the result of a signed arithmetic operation produced a negative result. This bit is also a general-purpose indicator of the state of the most significant bit (bit 7) of the accumulator.

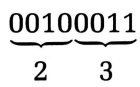
The 6502 can be programmed to test the condition (state) of each of these flags. Based on the results of these tests, the 6502 can "decide" which of two possible execution paths it must take. (The flags themselves are unaffected by these tests.)

The remaining usable bits in the processor status register are control bits. The IRQ disable bit (I) is used to "lock out" external interrupts to the 6502 when, for some reason, the microprocessor is not prepared to service an interrupt. The decimal mode

bit (D) controls whether the 6502 is to operate as a binary adder or decimal adder. In the binary mode, arithmetic operands are treated as 8-bit binary numbers. In the decimal mode, arithmetic operands are treated as two 4-bit BCD (binary coded decimal) digits "packed" into one 8-bit byte. For example, the 8-bit pattern for the value 23 is



in the binary mode and



in the decimal, or BCD, mode.

Memory

If the Apple's microprocessor had *thousands* of registers rather than just six, you would probably have ample space to store all the programs and numbers you could possibly use. In the absence of those thousands of registers, however, you will find yourself working instead with storage outside the microprocessor—the storage provided in the computer's *memory*. In fact, a great deal of your programming will involve transferring a number from memory into a register, performing one or more operations on that number (perhaps adding a value to it), then returning the result to memory. Since memory operations are such an integral part of programming, we should take a closer look at them.

In a 6502-based computer, such as the Apple II, memory can consist of up to 65,536 *different locations*, where each location is 8 bits (or 1 byte) long. (Computerists usually refer to memory capacity in units of K's, where K = 1024 bytes, so if someone says "I have a 48K Apple," you know that Apple contains 49,152 bytes of memory.) Each location in memory has a unique *address*, an identifying number from 0 to 65535 in decimal or 0 to FFFF in hexadecimal. You might imagine memory as a post office with 65,536 mail boxes, each labeled with a number. This post office has just one employee, named Mike Roprocessor, the only person with access to the boxes. Whenever you need to fetch a package (number) from a particular box, or deposit a package into a box, Mike will do it for you, but he must be told *which* box is to be accessed.

The simplest way to tell Mike which box to access is to tell him the box number (address) directly, for example, "Please give me the contents of box 100, Mike." However, sometimes you don't know which box holds your package, but you do have

some information as to how the correct box can be located. For instance, you might be picking up a package for a friend who lives six houses down the street from you. Although you don't know the friend's address, your own mail box is box 100, so you say, "Mike, I'd like to have the contents of the sixth box past box 100."

Simplistic as they are, these two scenarios are, in fact, valid illustrations of two different ways the 6502 microprocessor can access locations in the Apple II's memory. In addition, the 6502 has eleven other ways to access data, giving it *thirteen addressing modes* in all. In Table 1–1 you saw a list of the fifty-six mnemonics in the 6502 instruction set. These mnemonics tell the microprocessor which of fifty-six different operations to perform. In operations that involve data, the mnemonic tells the 6502 what to *do* with the data (add, subtract, store in memory, or whatever). Knowing now that most data are held in memory, you must have some means of telling the microprocessor where the data are located or, following an operation, where the result is to be stored.

You tell the microprocessor where to obtain data, or where to store data, by using one of the thirteen addressing modes in a special part of the instruction called the *operand field*. Therefore, most instructions that operate on data have not only a mnemonic, but an operand as well. For example, in the instruction

```
ADC  $400
```

ADC is the mnemonic and $400 is the operand. The mnemonic ADC tells the 6502 to add a data value to the contents of the accumulator. The operand $400 tells the 6502 that the data to be added are located in hexadecimal location 400. (The $ prefix informs the assembler that 400 is a hexadecimal value rather than a decimal value. More on this in Chapter 2.)

In the preceding ADC instruction, the address of the data is specified directly in the operand field. This mode, called *absolute addressing*, is one of the most straightforward of the 6502's thirteen addressing modes. However, the ADC instruction is not limited to absolute addressing. In fact, it can use any of eight different modes! For any particular instruction, the 6502 "knows" which addressing mode is being used based on the op-code it receives from memory. Instructions with just one possible addressing mode will have only one valid op-code, but instructions with several possible addressing modes (such as ADC) will have several different valid op-codes.

Because each instruction has only one mnemonic, the assembler generates the correct op-code based on the *format* of the operand. Therefore, each of the thirteen addressing modes must have its own unique format. Table 1–2 shows the formats for the example assembler used in this book, the Assembler/Editor offered by Apple Computer Inc.

Operand Addressing Modes

Following are descriptions of all thirteen operand addressing modes. Read these descriptions for general understanding, but don't attempt to memorize details. The details will come with practice as you perform the exercises in this book.

TABLE 1–2. Operands for 6502 Addressing Modes

Mode	Operand Format
Immediate	#data
Absolute	abs-addr
Zero Page	zpg-addr
Implied	
Absolute Indirect	(abs-addr)
Absolute Indexed, X	abs-addr,X
Absolute Indexed, Y	abs-addr,Y
Zero Page Indexed, X	zpg-addr,X
Zero Page Indexed, Y	zpg-addr,Y
Indexed Indirect	(zpg-addr,X)
Indirect Indexed	(zpg-addr),Y
Relative	abs-addr
Accumulator	A

Legend: data = an 8-bit data value.
abs-addr = a 16-bit absolute address (0 to 65535 decimal or 0 to FFFF hexadecimal).

zpg-addr = an 8-bit zero page address (0 to 255 decimal or 0 to FF hexadecimal).

Immediate addressing allows you to specify an 8-bit constant as an operand. For example, the instruction

```
LDA  #$3F
```

loads the hexadecimal value 3F (decimal 63) into the accumulator. Instructions using immediate addressing occupy 2 bytes in memory: an op-code byte followed by a data byte.

Absolute addressing allows you to directly address any of the 65,536 memory locations in the 6502's "address space." For example, the instruction

```
LDA  $400
```

loads the contents of memory location $400 (again, hexadecimal) into the accumulator. Instructions using absolute addressing occupy 3 bytes in memory; an op-code, followed by the low-order address byte and the high-order address byte. Note that the 6502 stores addresses in the opposite order you would expect, low byte then high byte rather than vice versa.

Zero page addressing is a form of absolute addressing in which the 6502 can access only the first 256 locations in memory, 0 through $FF (or 0 through 255 in decimal). Because the high-order byte of a zero page address is always zero, instructions using zero page addressing occupy only 2 bytes in memory. This 1-byte savings over absolute addressing means that zero page instructions can execute faster than absolute addressed instructions. The assembler will treat a two-digit hexadecimal oper-

and (or a decimal operand between 0 and 255) as a zero page address. Therefore, the instruction

```
LDA  $2A
```

loads the contents of memory location $2A into the accumulator.

With two exceptions, JMP and JSR, every instruction that can use absolute addressing can also use zero page addressing. Considering the savings in both storage space and execution time, you should use the zero page whenever possible to hold frequently accessed data or temporary data.

Roughly half of the 6502's instructions perform simple workmanlike tasks, such as setting or clearing a bit in the processor status register, incrementing or decrementing a register, or copying the contents of one register into another. These instructions need no operand—the 6502 receives sufficient information from the op-code alone—and employ what is called (appropriately) *implied addressing*, because the operation is implied in the op-code.

Following are some examples of instructions that use implied addressing:

Mnemonic	Description
CLC	Clear Carry Flag
DEX	Decrement the X Register
TAX	Transfer Accumulator to X Register

All implied addressing instructions occupy 1 byte in memory, which holds the op-code.

Absolute indirect addressing is used by only one instruction, *jump* (JMP). The JMP instruction loads the program counter with a new address at which the 6502 is to fetch its next instruction. JMP can use either absolute or absolute indirect addressing. With absolute addressing, the operand of the JMP instruction is the address that is to be put into the PC. With absolute indirect addressing, the operand of the JMP instruction is the address of the first of two memory locations that *contain* the new destination address.

An absolute indirect operand is specified by enclosing it in parentheses. For example,

```
JMP  ($203)
```

causes the program counter to be loaded with the low-order address contained in location $203 and the high-order address contained in location $204. If locations $203 and $204 contain the values $BC and $04, respectively, the PC will be loaded with the value $04BC, so the 6502 will fetch its next program instruction from location $04BC. Figure 1–3 illustrates this example.

As you can see, absolute indirect addressing permits the new address to be a

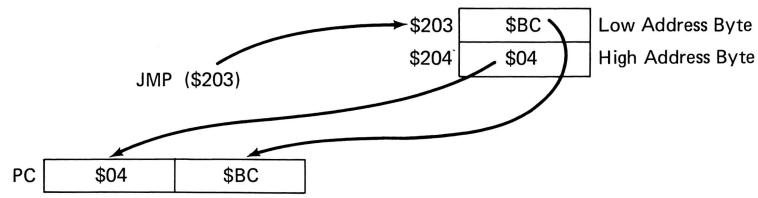


FIGURE 1–3. Absolute Indirect Addressing

variable quantity. For instance, an absolute indirect JMP instruction could be used in a system that services many users. If the address of user 1's program is in the indirect location ($203 in the preceding example), the 6502 will jump to user 1's program; if the address of user 2's program is in the indirect location, the 6502 will jump to user 2's program; and so on.

In *absolute indexed addressing*, the effective address of the operand is computed by adding the contents of the X or Y register to the absolute address in the operand, that is,

Effective address = absolute address + X

or

Effective address = absolute address + Y

All absolute indexed instructions occupy 3 bytes in memory. Absolute indexed operands are formed by attaching ,X or ,Y to the address. For example, if the X register contains $03, the instruction

LDA $12B4,X

loads the contents of memory location $12B7 (that is, $12B4 + $03) into the accumulator. Figure 1–4 illustrates this example.

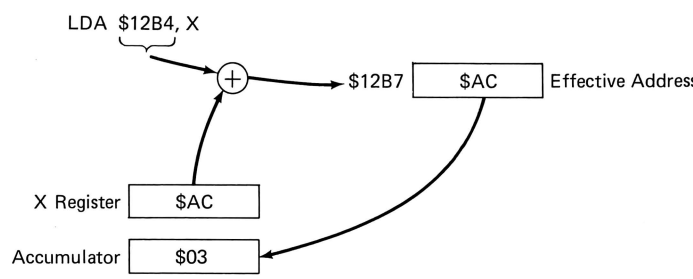


FIGURE 1–4. Absolute Indexed Addressing

Absolute indexed addressing is particularly useful for accessing data in a table. For this application you would use the starting address of the table as the operand address and use the index register (X or Y) to select the element in the table you want to access. If you increment the index after each access, you can access a series of consecutive elements in the table.

Zero page indexed addressing is identical to absolute indexed addressing except that the base address in the operand must be a zero page address (an address between 0 and $FF). All zero page indexed instructions are 2-byte instructions, 1 byte less than their absolute indexed counterparts.

Zero page indexed operands are formed by attaching a ,X or ,Y suffix to the address. For example, if the X register contains $03, the instruction

```
LDA   $2A,X
```

loads the contents of location $2D (that is, $2A + $03) into the accumulator.

Indexed indirect addressing is a combination of two addressing modes that have already been discussed, indexed addressing and indirect addressing. With indexed indirect addressing, the contents of the X register are added to a zero page operand to produce an indirect zero page address. The effective address of the data is contained in the two locations that start at this address (again, in low-byte/high-byte order).

Because the operand address is a zero page address, all indexed indirect instructions occupy only 2 bytes in memory: op-code and 1-byte address. Since the X register is 8 bits wide, it can be used to add an "offset" of up to 255 to the address in the instruction. The sum of these two terms (instruction address plus X register) must represent an address in page zero, however; that is, the effective address must be no greater than $FF.

The advantage of indexed indirect addressing is that *the entire 64K-byte memory space of the 6502 microprocessor can be accessed with a 2-byte instruction!* However, nothing comes for free, because the indexed indirect instructions take twice as long to execute as the zero page form of the same instruction and three times as long as the absolute form.

Indexed indirect operands are of the form (zpg-addr,X). For example, if the X register contains $4B, the instruction

```
LDA   ($2A,X)
```

causes the 6502 to compute an address of $75 (that is, $2A + $4B) and fetch the effective memory address from location $75 (low byte) and location $76. If locations $75 and $76 contain the values $B7 and $12, respectively, then the contents of location $12B7 will be loaded into the accumulator. Figure 1–5 illustrates this example.

The very nature of indexed indirect addressing, adding a displacement to a zero page base address to fetch an address, hints at a possible use—selecting one address from a table of addresses in page zero. Of course, since each address occupies two

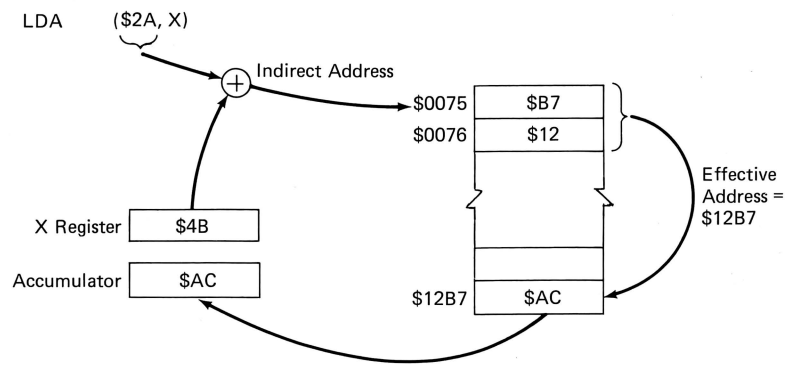


FIGURE 1–5. Indexed Indirect Addressing

memory locations, the contents of the X register must be doubled before being applied to the base address.

Indirect indexed addressing combines the same two addressing concepts as indexed indirect addressing (indexed addressing and indirect addressing), but applies them in reverse order. In the preceding mode, indexed indirect addressing, the index is added to the zero page operand *before* the indirect addressing is performed. This technique is called *preindexing*. In indirect indexed addressing, the index is added to the 16-bit memory address *after* the indirect addressing is performed. This technique is called *postindexing*.

Indirect indexed operands are of the form (zpg-addr),Y. For example, if the Y register contains $4B, the instruction

```
LDA   ($2A),Y
```

fetches its base address from zero page locations $2A (low address) and $2B (high address). If locations $2A and $2B contain the values $B7 and $12, respectively, the base address of the data table is $12B7. The value loaded into the accumulator is at location $12B7 + $4B, which means that the contents of location $1302 are loaded into the accumulator. Figure 1–6 illustrates this example.

Indirect indexed addressing is useful for accessing a certain element in one of a number of like structured data tables. This mode might perhaps be used in an instruction sequence shared by several users. Before executing the indirect indexed instruction, the user's calling program stores *his* data table's base address into the zero page operand location, so that the instruction ends up using the correct data table.

Relative addressing is just what the name implies; the effective address is specified relative to the contents of the program counter. Since the program counter contains the address of the next instruction to be executed, the effective address is some number of bytes higher in memory, or lower in memory, than the next instruction. Therefore, the effective address is computed by adding a positive or negative offset to the contents of the program counter. A positive offset will reference a location

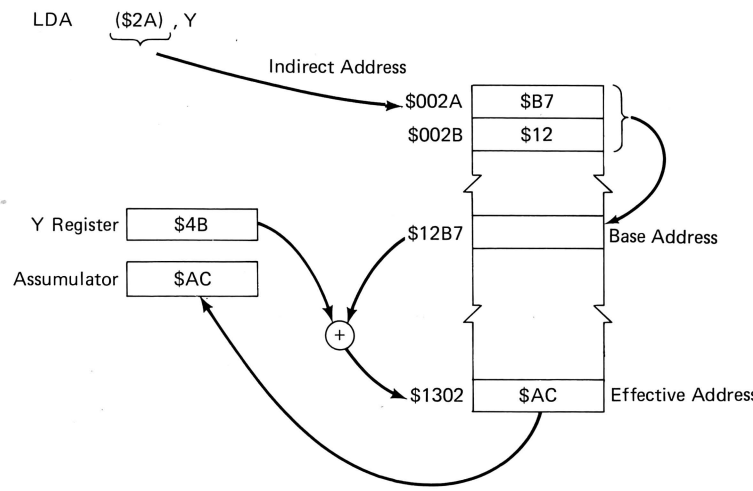


FIGURE 1–6. Indirect Indexed Addressing

following the next instruction (that is, higher in memory), whereas a negative offset will reference a location preceding the next instruction.

Relative addressing is used only by the 6502's eight *branch instructions*. The branch instructions cause program control to transfer forward or backward if a certain condition is met (for example, if a preceding arithmetic operation produced a zero result); otherwise execution proceeds to the next sequential instruction. For example, the branch on carry clear (BCC) instruction

```
        BCC   NEXT
NOTNXT  LDA   #$3F
        ..
        ..
NEXT    LDA   #0
```

will cause the 6502 to branch to the instruction at label NEXT (LDA #0) if the carry flag is clear (reset). If the carry flag is set, execution continues with the instruction at label NOTNXT (LDA #$3F).

All branch instructions occupy 2 bytes in memory, with the second byte containing the offset. Being an 8-bit value, the offset is limited to the range +127 bytes (forward) to –128 bytes (backward) from the program counter address. At any rate, branch instruction operands are almost always *labels* (as in the preceding example), so the assembler will calculate the offset for you. Branch instructions are covered in Chapter 4.

Accumulator addressing is limited to four instructions that shift or rotate the contents of the accumulator or a memory location one bit position to the right or left. If the operand of one of these instructions is A, the accumulator is shifted or rotated; if the operand has any other format, a memory location is shifted or rotated. Clearly,

TABLE 1–3 6502 Instruction Set Summary

INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM			IMPLIED			(IND, X)			(IND), Y			Z. PAGE, X			ABS. X			ABS. Y			RELATIVE			INDIRECT			Z. PAGE, Y			PROCESSOR STATUS CODES	
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7 6 5 4 3 2 1 0 N V • B D I Z C	MNEMONIC
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										N V • • • • Z C	ADC
AND	A∧M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										N • • • • • Z •	AND
ASL	C ← 7 6 5 4 3 2 1 0 ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													N • • • • • Z C	ASL
BCC	BRANCH ON C = 0 (2)																															90	2	2								• • • • • • • •	BCC
BCS	BRANCH ON C = 1 (2)																															B0	2	2								• • • • • • • •	BCS
BEQ	BRANCH ON Z = 1 (2)																															F0	2	2								• • • • • • • •	BEQ
BIT	A∧M				2C	4	3	24	3	2																																M_7 M_6 • • • • Z •	BIT
BMI	BRANCH ON N = 1 (2)																															30	2	2								• • • • • • • •	BMI
BNE	BRANCH ON Z = 0 (2)																															D0	2	2								• • • • • • • •	BNE
BPL	BRANCH ON N = 0 (2)																															10	2	2								• • • • • • • •	BPL
BRK	BREAK													00	7	1																									• • • 1 • 1 • •	BRK
BVC	BRANCH ON V = 0 (2)																															50	2	2								• • • • • • • •	BVC
BVS	BRANCH ON V = 1 (2)																															70	2	2								• • • • • • • •	BVS
CLC	0 → C													18	2	1																									• • • • • • • 0	CLC
CLD	0 → D													D8	2	1																									• • • • 0 • • •	CLD
CLI	0 → I													58	2	1																									• • • • • 0 • •	CLI
CLV	0 → V													B8	2	1																									• 0 • • • • •	CLV
CMP	A − M	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										N • • • • • Z C	CMP
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																															N • • • • • Z C	CPX
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																															N • • • • • Z C	CPY
DEC	M − 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													N • • • • • Z •	DEC
DEX	X − 1 → X													CA	2	1																									N • • • • • Z •	DEX
DEY	Y − 1 → Y													88	2	1																									N • • • • • Z •	DEY
EOR	A ∨ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										N • • • • • Z •	EOR
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													N • • • • • Z •	INC
INX	X + 1 → X													E8	2	1																									N • • • • • Z •	INX
INY	Y + 1 → Y													C8	2	1																									N • • • • • Z •	INY
JMP	JUMP TO NEW LOC				4C	3	3																												6C	5	3				• • • • • • • •	JMP
JSR	JUMP SUB				20	6	3																																			• • • • • • • •	JSR
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										N • • • • • Z •	LDA
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	N • • • • • Z •	LDX
LDY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													N • • • • • Z •	LDY
LSR	0 → 7 6 5 4 3 2 1 0 → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0 • • • • • Z C	LSR
NOP	NO OPERATION													EA	2	1																									• • • • • • • •	NOP
ORA	A ∨ M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										N • • • • • Z •	ORA
PHA	A → Ms S − 1 → S													48	3	1																									• • • • • • • •	PHA
PHP	P → Ms S − 1 → S													08	3	1																									• • • • • • • •	PHP
PLA	S + 1 → S Ms → A													68	4	1																									N • • • • • Z •	PLA
PLP	S + 1 → S Ms → P													28	4	1																									(RESTORED)	PLP
ROL	7 6 5 4 3 2 1 0 ← C				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													N • • • • • Z C	ROL
ROR	C → 7 6 5 4 3 2 1 0				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													N • • • • • Z C	ROR
RTI	RTRN INT													40	6	1																									(RESTORED)	RTI
RTS	RTRN SUB													60	6	1																									• • • • • • • •	RTS
SBC	A − M − $\overline{C}$ → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										N V • • • • Z (3)	SBC
SEC	1 → C													38	2	1																									• • • • • • • 1	SEC
SED	1 → D													F8	2	1																									• • • • 1 • • •	SED
SEI	1 → I													78	2	1																									• • • • • 1 • •	SEI
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										• • • • • • • •	STA
STX	X → M				8E	4	3	86	3	2																													96	4	2	• • • • • • • •	STX
STY	Y → M				8C	4	3	84	3	2													94	4	2																	• • • • • • • •	STY
TAX	A → X													AA	2	1																									N • • • • • Z •	TAX
TAY	A → Y													A8	2	1																									N • • • • • Z •	TAY
TSX	S → X													BA	2	1																									N • • • • • Z •	TSX
TXA	X → A													8A	2	1																									N • • • • • Z •	TXA
TXS	X → S													9A	2	1																									• • • • • • • •	TXS
TYA	Y → A													98	2	1																									N • • • • • Z •	TYA

(1) ADD 1 to "N" IF PAGE BOUNDARY IS CROSSED
(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE
(3) CARRY NOT = BORROW
(4) IF IN DECIMAL MODE, Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
Ms MEMORY PER STACK POINTER

+ ADD
− SUBTRACT
∧ AND
∨ OR
∀ EXCLUSIVE OR

M_7 MEMORY BIT 7
M_6 MEMORY BIT 6
n NO. CYCLES
NO. BYTES

Source: Courtesy of Rockwell International.

then, accumulator addressing is simply an implied type of addressing that is unique to the four shift-and-rotate instructions.

For example, the instruction

```
ASL  A
```

shifts the contents of the accumulator to the left by one bit position. Shift-and-rotate instructions are covered in Chapter 6.

Table 1–3 shows the allowable operand addressing modes for each of the 6502's fifty-six instructions, as well as the associated op-code (OP column), the number of machine cycles the instruction takes to execute (n column), and the number of bytes it occupies in memory (# column). It also shows how the processor status register bits are affected. Appendix A of the *Apple II Reference Manual* gives another version of Table 1–3, but minus the cycle count information.

The Monitor

One way to gain access to the 6502 microprocessor and the system memory is through the *Monitor*, the Apple II's main operating program. The Monitor provides keyboard commands that allow you to examine the contents of the 6502's registers and specific memory locations (and change those contents if desired), to execute a program in memory, and to perform a variety of other operations.

Monitor commands are described in Chapter 3 of the *Apple II Reference Manual* and a few of the more fundamental commands are summarized in Table 1–4. In this table, all addresses and data values are hexadecimal numbers.

To use any of the Monitor commands, your Apple II must be operating in the Monitor mode, indicated by an asterisk (*) prompt character on the screen. If your Apple's screen shows either the Integer BASIC prompt (>) or the Applesoft prompt ( ]), type in the command:

```
CALL -151
```

and press RETURN. This BASIC statement is actually a call to hexadecimal address $FF69, the Monitor's entry point in memory, but since BASIC does not accept hexadecimal numbers, a decimal equivalent was required.

With your Apple in the Monitor mode (the * prompt is on the screen), take a few minutes to learn the commands listed in Table 1–4. The first group of commands allow you to examine the contents of memory, as either individual locations or blocks of locations. The command **addr* displays the contents of a single location. For example, the command **0* produces the display:

```
0000- 4C
```

If you press RETURN at this time, you will get this display:

```
3C D4 4C 3A DB 00 00
```

TABLE 1–4 Fundamental Monitor Commands

Examining the Contents of Memory	
*addr	Displays the Contents of Location *addr.*
*addr *then* *(Press RETURN)	Displays the Contents of Up to Seven More Locations, Starting with *addr+1.* Each Subsequent RETURN Displays the Contents of the Next Eight Locations.
*addr1.addr2	Displays the Contents of Locations *addr1* Through *addr2.*
Changing the Contents of Memory	
*addr:val val . . .	Stores Values in Consecutive Memory Locations, Starting at *addr.*
*addr:val val . . . *then* *:val val . . .	Stores Values in Memory, Starting at the Next Location.
Examining the Contents of Registers	
*CTRL-E	Displays the Contents of the 6502's Registers in This Order: Accumulator (A), X Register (X), Y Register (Y), Processor Status Register (P), and Stack Pointer (S).
Changing the Contents of Registers	
*CTRL-E *then* :val val . . .	Loads Values into the Five Registers Listed Above. To Change Any Register Except A You Must Reenter the Contents of All Preceding Registers.
Running and Listing Programs	
*addrG	Executes a Machine Language Program, Starting at *addr.*
*addrL	Disassembles and Displays Twenty Instructions, Starting at *addr.* Each Subsequent *L Displays 20 More Instructions.

which are the contents of locations 1 through 7. This RETURN always displays locations up to addresses ending in 7 or F (such as 0007, 000F, 1007, 100F, and so on). Subsequent RETURNs display the next block of eight locations.

To examine more than eight locations at a time, you can use the command *addr1.addr2, where addr1 and addr2 are the first and last locations to be displayed,

respectively. For example,

```
*1.B
```

produces this display:

```
0001- 3C D4 4C 3A DB 00 00
0008- 00 00 4C 99
```

The next group of commands allows you to change the contents of memory by storing new values into the selected locations. For example, to clear locations 1 through 5 to zero, you type

```
*1:0 0 0 0 0
```

After the first change command, you can change additional locations by entering just the colon. For example, to clear locations 6 through A (remember, all Monitor values are hexadecimal), enter

```
*:0 0 0 0 0
```

The 6502's programmable registers can also be examined and altered with Monitor commands. To examine their current values, enter CTRL-E, then press RETURN. The resulting display looks like this:

```
A=98 X=84 Y=D8 P=00 S=B7
```

To change one or more of these register values, type a colon (:) followed by one to five hexadecimal values in the range 0 to FF. The Monitor doesn't permit you to skip over registers, however, so you'll have to reenter any that remain unchanged. For example, to change the Y register contents to $FF, enter

```
*:98 84 FF
```

and press RETURN. Displaying the registers again with CTRL-E should show

```
A=98 X=84 Y=FF P=00 S=B7
```

The Monitor's execute command, *addrG, is probably the most important of all because it allows you to run a program. For example, to run a program that starts at location $300 in memory, enter

```
*300G
```

Since you haven't yet stored a program in memory, there's no way to test this command just yet, but you will use it in the exercise at the end of this chapter and in every subsequent chapter.

The last command in Table 1–4 (**addrL*) takes a series of consecutive bytes in memory and *disassembles* them into twenty instructions. That is, the Monitor starts at the specified address (addr) and assumes that the contents of that byte represent the op-code of an instruction. If that byte *does* hold one of the legal op-codes (a value appearing in an OP column in Table 1–3), the Monitor displays that op-code's mnemonic abbreviation on the screen. Further, if the op-code represents an instruction having an operand, the Monitor converts the next two or three bytes into an operand and displays it with a format similar to those encountered in Table 1–2. If the presumed op-code byte does *not* represent a legal op-code, the Monitor displays ??? as the mnemonic.

This process continues until twenty lines are displayed on the screen. Each display line shows an address, one to three 2-digit hexadecimal values, a mnemonic (or ???) and, if appropriate, an operand. This disassemble command is very useful for "debugging" programs because it shows you exactly how a program is stored in memory, as opposed to how you *think* it's stored in memory.

Using only the Monitor commands in Table 1–4, it is entirely possible to enter, execute, and debug programs without an assembler. This type of programming is called *machine language programming* because you provide numeric values the "machine" (microprocessor) can interpret directly. You are, in effect, speaking the machine's language! In the upcoming exercise you will write and execute a short machine language program.

Load, Store, and Break Instructions

Of the fifty-six instructions in the 6502's repertoire, the most basic are those that transfer data between the microprocessor and memory. All such transfers involve the three general-purpose registers: the accumulator, the X register, and the Y register. The instructions that transfer memory data into a register are called *load instructions*. The instructions that transfer register data into memory are called *store instructions*.

There are three load instructions and three store instructions:

Instruction	Description
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory

All six transfers are "nondestructive"; loads do not alter memory and stores do not alter registers.

Table 1–3 shows the allowable addressing modes for each of these instructions. For example, LDA can use any of eight addressing modes, but LDX and LDY are limited to four and five modes, respectively. This tells you that indexed indirect instruction

```
LDA   ($03,X)
```

is a legal instruction, but

```
LDX   ($03,X)   and   LDY   ($03,X)
```

are illegal.

Further, the processor status codes column of Table 1–3 shows how the load and store instructions affect the processor status register's five flags. Each of the load instructions can affect two flags, negative (N) and zero (Z). These flags indicate whether the loaded value is negative (N=1) or nonnegative (N=0) and whether it is zero (Z=1) or nonzero (Z=0). Store instructions do not affect the flags.

In Table 1–4 you saw a Monitor command that starts a program executing. How do you *stop* the execution? The normal way is with the instruction:

Instruction	Description
BRK	Break

When the Apple II's 6502 microprocessor encounters a BRK instruction in a program, it returns control back to the Monitor and displays the contents of six registers: program counter, accumulator (A), X register (X), Y register (Y), processor status register (P), and stack pointer (S). Due to the way BRK operates, however, the program counter contains the address of the location that lies two bytes past BRK.

For example, if a program starts at location $300 in memory and a BRK instruction is held in location $320, when you execute the program the following sequence may appear on the screen:

```
*300G
0322-    A=98 X=44 Y=00 P=30 S=F0
```

Incidentally, BRK's op-code is the easiest to remember; it is simply 00.

Because BRK is the standard program terminator, *every program you write for the exercises in this book should end with BRK,* unless you are instructed otherwise.

EXERCISES

In this exercise you will enter a machine language program into memory and execute it. By doing so you will gain experience in using the Monitor commands.

1. Turn on the Apple II power, then call the Monitor (CALL –151). What does the accumulator contain in this "initial" state?

   A =

2. Which Monitor command did you use to find out what the accumulator contained at this point?

3. What value is contained in location $10?

4. Which Monitor command did you use to find out location $10 contains?

5. Following is a three-instruction program that loads the value $AC into the accumulator and then stores the contents of the accumulator in location $10. This program is written in assembly language form (using mnemonics and symbolic operands). Since you aren't yet using an assembler, you must convert each mnemonic and operand into a two-digit hexadecimal value that can be entered into memory, then executed via the Monitor.
   a. For each of the three instructions in the following table, look up the hexadecimal op-code in Table 1–3 and record it (along with the hexadecimal operand value) in the space provided.

Instruction	Hex. Values
LDA #$AC	
STA $10	
BRK	

   b. Which addressing mode is used in the load instruction?

   c. Which addressing mode is used in the store instruction?

   d. Which addressing mode is used in the break instruction?

6. Enter the five hexadecimal values from step 5a into memory, starting at location $300.
   Which Monitor command did you use to do this?

7. Disassemble your program to ensure that you entered it correctly. The mnemonics and operands should match those given in the left-hand column of the table in step 5a.
   Which Monitor command did you use to disassemble the program?

8. Execute the program from its starting point. Which Monitor command did you use to do this?

9. What are the final contents of the program counter, accumulator, and location $10?

   PC = A = ($10) =

CHAPTER TWO

Using an Assembler

OBJECT

To develop programs in assembly language using an editor and an assembler.

NEW INSTRUCTIONS

This chapter describes the addition-related instructions ADC, CLC, SED, and CLD.

PREPARATION

Study the documentation supplied with the assembler/editor software package you will use with your Apple II.

DISCUSSION

In Chapter 1 you developed a short program to load a value into a memory location, using various Monitor commands. This required you to look up hexadecimal op-codes in a table and enter those values, along with hexadecimal operand values, into memory. In doing so, you programmed in *machine language*, that is, you provided the 6502 microprocessor with numeric values it could interpret directly. Although machine language programming ultimately gets the job done, the time spent looking up op-codes is grossly inefficient and highly error prone for all but the shortest programs.

In Chapter 1 you were introduced to a more advanced programming technique, one in which three-letter abbreviations, called *mnemonics*, could be substituted for numeric op-codes and symbolic operands could be used to provide addressing information. Since the 6502 can only "understand" numeric op-codes and operands, this symbolic approach requires a special program to translate the symbols into numeric (machine code) form. The program that performs this translation is called an *assembler*, so this form of programming is called *assembly language programming*.

In standard computer terminology, the symbolic form of a program is called *source*

code and the numeric, or machine code, form of a program is called *object code*. Therefore, the assembler translates source code into object code.

Most assembler software packages include a second program, called an *editor*, with which you prepare source code for the assembler. The editor is a text-processing program that lets you enter text into memory, add and delete characters and lines, transfer text to and from disk, and so on. However, being a *text*-processing program, the editor makes no qualitative "decisions" about what's typed into it. You could type a poem and a shopping list into memory and the editor would accept them!

To properly use the editor and assembler to develop assembly language programs you will want to know (1) what format the source code must have, (2) how to use the editor, and (3) how to use the assembler. The following sections address each of these topics for one particular software package: the *Apple 6502 Assembler/Editor*, which is part of the *DOS Tool Kit* developed by Apple Computer Inc. If you have some other assembler/editor package, you should still be able to apply the principles given here, with minor modifications.

Format of Source Code

Before you can use the editor to type source code (that is, assembler input) into memory, you must know *what* information the assembler will accept and *how* this information must be formatted. From Chapter 1 you already know that an assembly language instruction consists of two things: a mnemonic and an operand. Besides these, every assembler will permit an instruction (mnemonic plus operand) to be preceded by a *label* and followed by a *comment*.

Therefore, an instruction line can have the format:

(label) mnemonic operand (comment)

where label and comment are optional; you can include either or both fields, as you wish.

The *label field* is used to assign a name to an instruction, so that it can be referenced by other instructions in the program. A label must begin with a letter (A through Z) and must contain only letters, numerals, and periods. The assembler will accept labels of up to 250 characters, but you should make it a practice to limit them to eight characters or less. Labels are always optional, but if used they must start in the leftmost position on the line. Don't worry about labels just yet; you will learn how to use them in Chapter 4.

The *mnemonic field*, required in every instruction line, holds one of the fifty-six mnemonics from Table 1–1. If a label is used, the mnemonic must be separated from the label by at least one blank space. In the absence of a label, the mnemonic must start in the second character position on a line.

If used, the *operand field* must hold an operand formatted according to the addressing modes listed in Table 1–1. An operand must be separated from a mnemonic by at least one blank space. Unlike the Monitor, the assembler allows you to specify data values and addresses in decimal form as well as hexadecimal form. To distinguish

them, *hexadecimal values must be preceded by a dollar sign ($), whereas decimal values require no prefix.* For example, the instruction

```
LDA   #10
```

loads the decimal value 10 into the accumulator, whereas the instruction

```
LDA   #$10
```

loads the hexadecimal value 10 (that is, decimal 16) into the accumulator. Be sure to include the $ prefix if you are entering a hexadecimal number; it may save you considerable debugging time later.

The optional *comment field* allows you to provide a short phrase describing what the program is doing at that particular time, to make the program easier to follow. The comment field is ignored by the assembler, but is included if the program is listed on the screen or on a printer. If used, a comment must be preceded by a semicolon (;) and must be separated from the preceding field by at least one blank space.

Comments may also be used apart from instructions to provide an entire line of description. To do this, place an *asterisk* (*) at the leftmost character position on a line, which tells the assembler to ignore the entire line.

Here is a line that uses all four fields:

```
CLEARA   LDA   #0   ;Clear the Accumulator
```

Note that the comment includes lower case letters. Although they are not available from the standard Apple II keyboard, comments within this book are printed in lower case to make them easier to read.

In addition to supplying the assembler with the program instructions to be assembled, you must also tell the assembler where the object code is to be stored in memory. This is done with a special instruction to the assembler, called an *assembler directive.* There are a number of assembler directives, but the one that specifies where the object code is to be stored in memory is the *origin directive,* abbreviated ORG.

Every source code program you enter into the Apple, using the editor, must be preceded by an origin directive of the form

```
ORG   addr
```

For example, if you wish to have a program stored in memory starting at location $300, the source code must be preceded by ORG $300. Like an instruction mnemonic, ORG must start in the second position on a line and must be separated from its operand by at least one blank space.

How to Use the Editor

Now that you know what kinds of information the assembler expects in a source code program, it's time to learn the mechanics of entering this information, using the editor. Since it's always easier to learn by doing rather than just reading, you should perform the steps in this section as they are described.

There are two ways to get the Assembler/Editor program running in your Apple II:

1. If DOS has not been booted, place the Assembler/Editor diskette into disk unit 1 and enter

```
RUN EDASM
```

   This will cause an I.D. message to be displayed. Press RETURN in response to that message.
2. If DOS is already booted, place the Assembler/Editor diskette into disk unit 1 and enter

```
BRUN EDASM.OBJ
```

Both methods will display a "hello" message in the center of the screen and a colon (:) and flashing underline in the lower left corner. The colon is the editor's command mode prompt, which tells you that the Apple is waiting for an editor command. The flashing underline is the editor's cursor character.

Table 2–1 shows the editor commands you will use to prepare source code text for the assembler. These are all single-key commands; only the first letter is required, the rest are optional. Each command takes one or two additional parameters; some are optional (those in brackets), others are required.

The *add command* is the first you will use because it lets you enter a source code program into the editor's *edit file,* the portion of memory set aside for source code text. The editor automatically numbers each line of text you type in, for subsequent reference by other editing commands.

Since you've already developed a program in Chapter 1, you can use an assembly language version of that program as initial input to the edit file. To do this, you must type in the instructions one per line, and precede each mnemonic with a space (since you aren't using labels here). Here is what you should type in:

```
:A (Press RETURN here) ("Ø" means "space" on the lines below)
  1 ØLDA #$AC ;LOAD $AC INTO ACCUM.
  2 ØSTA $10 ; AND INTO LOCATION $10
  3 ØBRK  ;RETURN TO MONITOR
  4 (Press CTRL-D, then RETURN)
:
```

The *change command* allows you to change a series of characters (a "string") within the edit file. This command takes two parameters, an optional *Rangelist* and a

TABLE 2–1. Editing Commands

ADD Format: Add [Linenum]

Used to Add Text to the Edit File, Following Linenum. If *Linenum* is Omitted, Text Is Appended to the End of the Edit File. This Form Is Also Used to Establish an Edit File.

The Add Command Puts the Editor in the Input Mode, in Which All Normal APPLE II Editing Functions Are Available. These Include the Escape Sequences Listed in Table 1–1.

To Exit the Input Mode, Enter CTRL-D or CTRL-Q, Followed by RETURN.

CHANGE Format: Change [Rangelist] Chgstring

Lets You Substitute a New String for Some or All Occurrences of an Old String, Based on Your Response to the Prompt.

```
ALL OR SOME? (A/S)
```

With S, to Reinstate (or "Unchange") Any Particular Occurrence of an Old String, press ESC.

If *Rangelist* Is Omitted, the Entire Edit File is Available for String Substitutions.

DELETE Format: Delete Rangelist

Deletes All Lines Within the Specified *Rangelist.*

EDIT Format: Edit [Rangelist] [Dstring]

Places Editor in the Edit Mode, Giving You Complete Character-by-Character Editing Capability. One Line at a Time is Edited; Use the Right Arrow Key to Advance the Cursor and the Left Arrow Key to Backspace the Cursor. See Table 2–2 for Edit Mode Commands.

FIND Format: Find [Rangelist] [Dstring]

Locates and Lists All Lines Within *Rangelist* that Contain an Occurrence of *Dstring.* If *Dstring* Is Omitted, All Lines Within *Rangelist* Are Listed.

INSERT Format: Insert Linenum

Puts the Editor into Input Mode (see Add Command) and Inserts All New Lines Ahead of *Linenum.* To Exit the Input Mode, Enter CTRL-D or CTRL-Q, Followed by RETURN.

LIST Format: List [Rangelist]

Lists the Lines Within *Rangelist* on the Screen and Precedes Each Line with a Line Number.

PRINT Format: Print [Rangelist]

Same as List Command, but Line Numbers Are Not Displayed.

REPLACE Format: Replace Range

Deletes the *Range* of Lines and Places the Editor in the Input Mode (see Add Command).

required *Chgstring*. Rangelist is used to specify the groups of line numbers that are to be considered for changing. For example, if the change command is to operate on a large program, but must only affect lines 1 through 3, 16 through 20, and 30 through 33, the Rangelist would have the format

```
1–3,16–20,30–33
```

The second parameter, Chgstring, comprises two strings (an "old string" and a "new string") separated by a delimiting character, in this format:

```
⟨delimiter⟩⟨oldstring⟩⟨delimiter⟩⟨newstring⟩⟨delimiter⟩
```

The delimiter can be any character: a period, a comma, a dollar sign, etc.

The change command gives you the option of changing all occurrences of the old string, or just some occurrences, based on your response to the prompt

```
ALL OR SAME? (A/S)?
```

If you respond with S, each line will be displayed in its changed form. If the change is valid on that line, press RETURN; if you want to retain the old string, press ESC.

Use the change command to alter the comment for line 2 of the program, by shortening "LOCATION" to "LOC." as follows:

```
:C 2 $ATION$.$
```

Respond to the prompt by typing A (although S would do equally well in this instance). At this point, the editor displays the changed line but seems to scatter the text all over the screen! In doing this, it has actually formatted your instruction into four separate fields: label (missing from your instructions), mnemonic, operand, and comment.

The *edit command* provides a set of control characters (Table 2–2) that allow you to make corrections to the edit file. The edit command also takes two optional parameters, *Rangelist* and *Dstring*, which can limit the editing to groups of lines and/or lines containing a certain string.

With edit's control characters you can move the cursor; delete, insert, and find a character; truncate a line; and cancel existing corrections on a line. (Although not listed in the table, to change a character at the cursor position, you just type over it.) When editing is complete, you can return to command mode with either RETURN or CTRL-X.

Two of these functions, CTRL-I and CTRL-F, must be immediately followed with another key, which specifies the character to be inserted or located, respectively. *Do not* type a space between CTRL-I/CTRL-F and the specifier key.

The remaining editor commands are self-explanatory, so you should have no trouble using them. There *is*, however, one more line that needs to be added to the edit file before it can be assembled: You need to add an ORG directive, to tell the assembler

TABLE 2–2. Edit Mode Control Characters

Character	Action
→	Moves Cursor One Position to the Right.
←	Moves Cursor One Position to the Left.
RETURN	Accepts Line as Displayed and Moves to Next Line. If the Line Is the Last in Rangelist, RETURN Causes an Exit to Command Mode.
CTRL-D	Deletes Character at Current Cursor Position, Shortening the Line by One Character.
CTRL-I char	Inserts *char* Ahead of the Current Cursor Position.
CTRL-F char	Finds the Next Occurrence of *char* on the Current Line.
CTRL-T	Truncates the Line, Starting at the Current Cursor Position.
CTRL-R	Replaces the Edited Line with the Original (Unedited) Line.
CTRL-X	Exits Edit Mode, Canceling All Editing Corrections on the Current Line.

where the object code is to be stored in memory. Using the insert command, you should insert

```
bORG $300 ;START PROGRAM AT LOC. $300
```

as a new line 1 in the edit file.

After making this insertion, a *list* of the complete source code program should look like this:

```
:LIST
    1           ORG  $300      ;START PR
      OGRAM AT LOC. $300
    2           LDA  #$AC      ;LOAD $AC
       INTO ACCUM.
    3           STA  $10       ; AND INT
      O LOC. $10
    4           BRK            ;RETURN T
      O MONITOR
```

SAVEing the Source Text on Disk

The assembler requires the source code to be on disk rather than in memory, so you must now write the edit file onto a diskette. To do this, replace the Assembler/Editor diskette with an initialized, writeable diskette (see Chapter 2 of the DOS Manual), then type in this editor command:

```
:SAVE CH2EX
```

and press RETURN. The four-line edit file will be saved under the file name CH2EX (for Chapter 2 example). Verify that this took place with :CATALOG.

Incidentally, the saved edit file is still in memory at this point, in case you still want to edit it.

How to Use the Assembler

With the final source code on diskette, you can now begin the process of converting it—that is, *assembling* it—into executable 6502 object code. As mentioned earlier, this task is performed by the *assembler*.

The editor and assembler cannot both be in memory at the same time, so you must "call" the assembler program into memory. The proper procedure is as follows:

1. Replace the source code diskette with the Assembler/Editor diskette.
2. Type in the command:

   ```
   :ASM CH2EX
   ```

   and press RETURN.
3. When the assembler has been loaded into memory, the display will show the message:

   ```
   PRESS ANY KEY TO CONTINUE_
   ```

   This tells you the assembler is waiting for you to put the source code diskette back into disk unit 1.
4. Replace the Assembler/Editor diskette with the source code diskette, then press some letter key or the space bar.
5. If you did everything correctly, the display should show the name of the object code file (it has the "default" name CH2EX.OBJO here because you did not give it another name in step 2), a disassembly of the program, and the message:

   ```
   *** SUCCESSFUL ASSEMBLY: NO ERRORS
   ```

   The last line is the prompt:

   ```
   PRESS ANY KEY TO CONTINUE_
   ```

6. This final message indicates that the assembler has finished its task, so you can reload the editor. Replace your program diskette with the Assembler/Editor diskette and, once again, press any key.

7. When you see the editor's command mode prompt (the colon), type

```
:END
```

and press RETURN. This will put you back into BASIC.

8. Now, to *execute* the program, put the source program back into the drive and enter

```
BRUN CH2EX.OBJ0
```

When the 6502 encounters the final instruction, BRK, it displays the register values (note A = AC) and enters the Monitor.

You have now completed all steps needed to prepare a source program using the editor and assemble that program using the assembler.

Usable Memory in the Apple II

The Apple II's 65,536-byte *address space* contains three different types of memory: read/write memory (or RAM), input/output (I/O) locations, and read-only memory (or ROM). As a programmer, it is important to know where these different types of memory are located, that is, which addresses they occupy, so you know where you can store programs and data.

Figure 2–1 shows a *memory map* of a 48K-byte Apple II system. RAM memory extends from location 0 to location $BFFF; I/O occupies $C000 to $CFFF; ROM occupies $D000 to $FFFF.

As you can see, "free RAM," in which you can store your programs, occupies the following address ranges:

- $300 through $3CF
- $C00 through $1FFF
- $6000 through $95FE

In a 32K-byte system, the latter range is occupied by the disk operating system (DOS) and is therefore unavailable to you. In this book, most programs will be stored into the free RAM space between locations $300 and $3CF.

Some, but not all, locations in page zero are also free for storing temporary data values. According to Table 18 in Chapter 4 of the *Apple II Reference Manual*, these free locations are:

- 0 through $1F
- $4A through $4D
- $56 through $FF

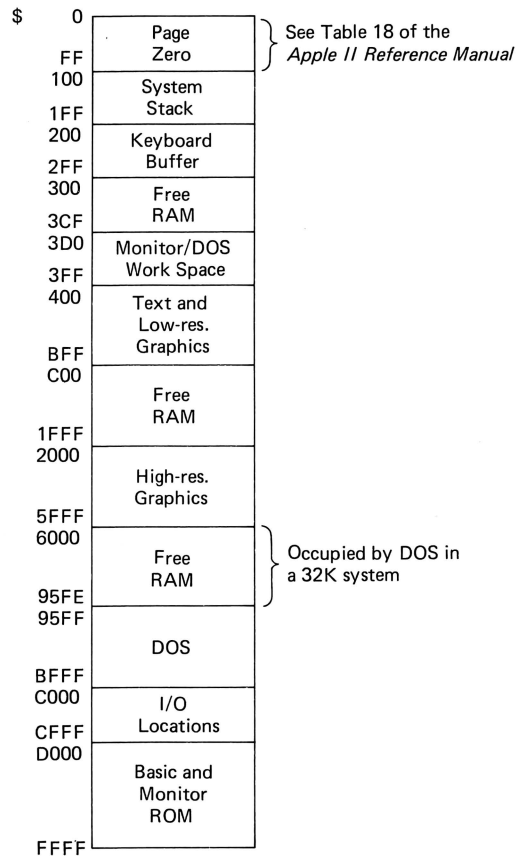


FIGURE 2–1. APPLE II Memory Map (48K System)

Addition Operations

In the exercise at the end of this chapter you will develop a program to add two numbers; therefore, you must be familiar with how additions are performed in assembly language.

Because the 6502 is an *8-bit* microprocessor it can only add two 8-bit values in a single operation. If the numbers you wish to add are too large to be contained in a single byte, you will need to perform two or more separate additions. Specifically, you must add the two lower order bytes, then add the next two bytes, and so on. Moreover, all additions except the first one must include any *carry* produced by the preceding addition. Figure 2–2 illustrates how two 3-byte operands in memory are added. (Note that a carry out of the third addition indicates an error condition; the sum was too large to be contained in 3 bytes.)

Looking at Figure 2–2, you might expect the 6502 to have two separate add instructions, one that includes carry in the addition and another that omits carry. However, the 6502 has only one add instruction, *add memory to accumulator with carry (ADC)*. You will see how the 6502 handles the no-carry case in a moment.

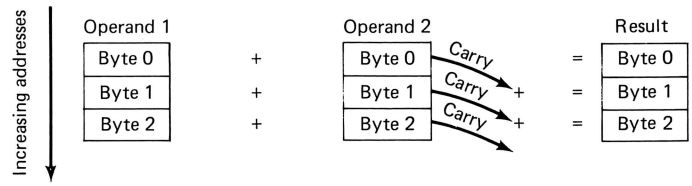


FIGURE 2–2. Adding Two Multibyte Numbers

The ADC instruction adds the contents of a specified memory location (or an immediate operand) and the 1-bit contents of the status register's carry flag to the accumulator. The general equation is

$$A = A + M + C$$

The ADC instruction can be used to add three types of integer numbers: (1) unsigned binary numbers, (2) signed binary numbers, and (3) binary coded decimal (BCD) numbers.

ADC affects four flags in the processor status register (see Figure 2–3):

- The carry (C) flag is set if the sum of a binary addition exceeds decimal 255 ($FF) or if the sum of a BCD addition exceeds decimal 99 ($99); otherwise it is reset.
- The zero (Z) flag is set if the sum is zero; otherwise it is reset.
- The negative (N) flag is set if bit 7 of the sum is a logic one; otherwise it is reset. The state of the N flag is meaningful only if signed numbers are being added, in which case it indicates whether the result is negative (N = 1) or positive (N = 0).
- The overflow (V) flag is set if two like signed numbers (both positive or both negative) are added and the sum exceeds +127 ($7F) or –128 ($80), which causes bit 7 of the accumulator to be changed; otherwise V is reset.

6502 Instructions Used for Addition

In all, four instructions are used during addition operations:

Instruction	Description
ADC	Add Memory to Accumulator with Carry
CLC	Clear Carry Flag
SED	Set Decimal Mode
CLD	Clear Decimal Mode

You're already familiar with the first instruction, ADC. The second instruction, CLC, is the one needed to eliminate carry when the first two bytes of a number are added.

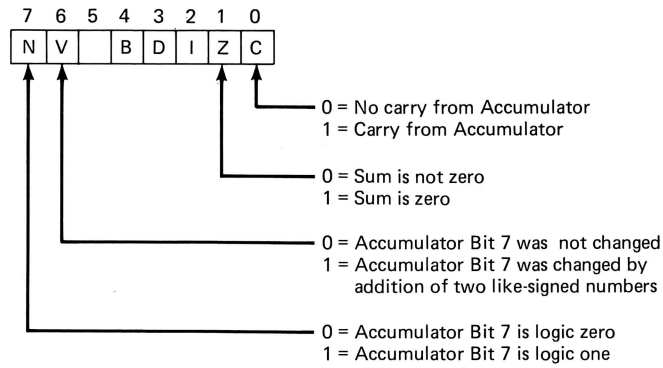

7 6 5 4 3 2 1 0
N V B D I Z C
0 = No carry from Accumulator
1 = Carry from Accumulator
0 = Sum is not zero
1 = Sum is zero
0 = Accumulator Bit 7 was not changed
1 = Accumulator Bit 7 was changed by addition of two like-signed numbers
0 = Accumulator Bit 7 is logic zero
1 = Accumulator Bit 7 is logic one

FIGURE 2–3. Status Flags for Addition

Thus, *every add sequence must be preceded by a CLC instruction*. The decimal mode instructions put the 6502's internal adder into the decimal or binary mode, respectively.

The Flowchart, A Program's Road Map

Just as you wouldn't think of starting a long trip without consulting a road map, you shouldn't consider writing a complex computer program without first drawing its "road map," called a *flowchart*. A flowchart is a diagram showing the operations the program must perform and the sequence in which they should be executed. Like a road map, a flowchart does not document every minute detail of a program, but instead gives a general "plan of action" upon which you can develop the actual instructions.

Flowcharts are made up of boxes, diamonds, and other shapes, interconnected with arrows. Figure 2–4 shows the shapes you should use for constructing the flowcharts

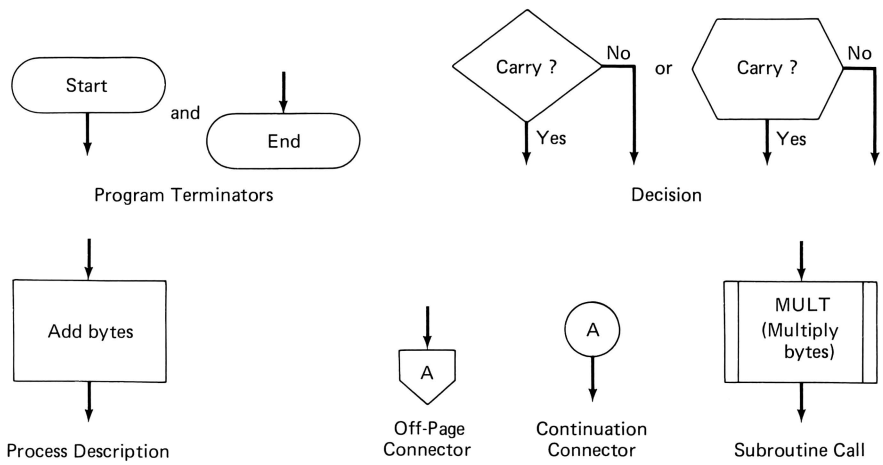


FIGURE 2–4. Flowchart Shapes

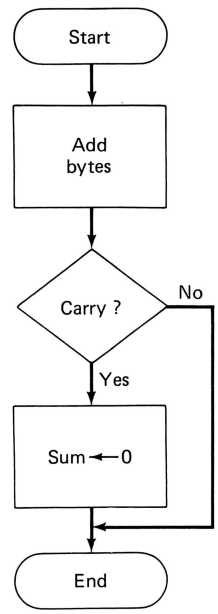


FIGURE 2–5. A Sample Flowchart

requested in the exercises in this book. Figure 2–5 shows a flowchart that might be prepared for a program that adds 2 bytes and zeroes the result if the addition produces a carry (that is, if the sum is too large to be contained in a single byte). Don't worry about the instructions that make the "carry/no carry" test; you will learn about such instructions in Chapter 4. The final step, end, is normally a BRK instruction, which sends the 6502 back into the Monitor.

You should make it a habit to draw a flowchart before writing all but the simplest programs. A good flowchart is not only the key to developing a program that will run correctly the first time, but is also a valuable "debugging" aid if errors occur.

EXERCISES

1. Develop a program to add the *unsigned integer contents* of memory locations $00 and $01, storing the sum in location $02.
   a. Draw the flowchart below.

   b. Write the program code below.

   c. Run* your program for the three cases listed below and complete the table.

$00		$01		$02		Carry?
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	(Yes/No)
143	8F	131	83	274		
46	2E	65	41	111		
59	3B	197	C5	256		

*Throughout this book, the word run presupposes that you've already entered the source code using the editor and have assembled it using the assembler, then read it into memory.

2. a. Write a program to add the *signed* integer contents of memory locations $00 and $01, storing the sum in location $02.

Will the flowchart and/or program instructions differ from those in step 1? If so, describe the differences below.

b. Run your program for the cases listed below and complete the table.

$00		$01		$02		Overflow?
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	(Yes/No)
106	6A	89	59	195		
127	7F	3	03	130		
32	20	–3	FD	29		
–90	A6	–62	C2	–152		

3. a. Develop a program to add two 16-bit (2-byte) signed integers, one stored in locations $00 and $01, the other stored in locations $02 and $03. Return the sum in locations $04 and $05.
Store numbers with the low-order byte in the lower addressed location. For example, locations $00 and $01 hold the least significant byte (LSB) and most significant byte (MSB) of the first operand.

$00	LSB_A
$01	MSB_A
$02	LSB_B
$03	MSB_B
$04	LSB_{sum}
$05	MSB_{sum}

Flowchart the problem in the space below.

b. Write the program code below.

c. Run your program for these additions:

$2563 + $41AB
$83C7 + $F1B4

and complete the table below.

Addend #1		Addend #2		Sum		Overflow?
$00	$01	$02	$03	$04	$05	(Yes/No)

CHAPTER THREE

Subtraction and Logical Operations

OBJECT

To perform subtraction operations on signed and unsigned numbers, and to use the AND, OR, and exclusive-OR logical operations.

NEW INSTRUCTIONS

This chapter describes the subtraction-related instructions SBC and SEC and the logical instructions AND, ORA, and EOR.

DISCUSSION

Subtraction Operations

The 6502 instruction set contains only one subtract instruction, *subtract memory from accumulator with carry (SBC)*. This instruction subtracts the contents of the specified memory location (or the second byte of the instruction, if immediate addressing is used) and the contents of the status register's carry bit (in inverted form) from the accumulator.

Why is carry used in its *inverted* form? Because for subtraction, carry represents a borrow; a final carry = 1 indicates absence of borrow, carry = 0 indicates that the subtraction has produced a borrow. Although carry must be initialized to 0 for addition operations, it must be initialized to 1 for subtraction operations.

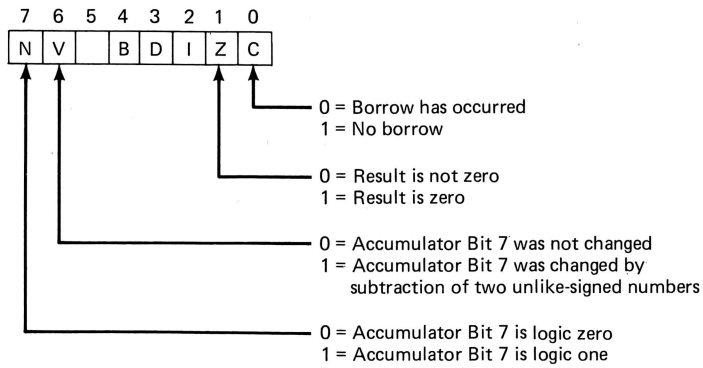

7 6 5 4 3 2 1 0
N V B D I Z C
0 = Borrow has occurred
1 = No borrow
0 = Result is not zero
1 = Result is zero
0 = Accumulator Bit 7 was not changed
1 = Accumulator Bit 7 was changed by
subtraction of two unlike-signed numbers
0 = Accumulator Bit 7 is logic zero
1 = Accumulator Bit 7 is logic one

FIGURE 3–1. Status Flags for Subtraction

Status Flags for Subtraction Operations

The SBC instruction affects four flags in the processor status register (see Figure 3–1):

- The carry (C) flag is set if the result is positive or zero, and is reset if the result is negative (indicating a borrow).
- The zero (Z) flag is set if the result is zero; otherwise it is reset. Note that if carry and zero are both set, the result is zero. If carry is set and zero is reset, the result is positive.
- The negative (N) flag is set if bit 7 of the result is a logic one; otherwise it is reset. The state of the N flag is meaningful only if signed numbers are being subtracted, in which case it indicates whether the result in negative (N = 1) or positive (N = 0).
- The overflow (V) flag is set if two unlike signed numbers are subtracted and the result exceeds $+127_{10}$ or -128_{10}, which causes bit 7 of the accumulator to be changed.

6502 Instructions Used for Subtraction

You will use the following new 6502 instructions in the subtraction portion of this exercise:

Instruction	Description
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag

The SBC instruction subtracts a specified memory location and the contents of the status register's carry flag from the accumulator. The inclusion of carry is convenient in multibyte subtractions, but it also forces you to set *carry (using SEC) before the first SBC instruction is executed.*

TABLE 3–1. Logical AND Operation

Memory Bit	Accumulator Bit	Result in Accumulator
0	0	0
0	1	0
1	0	0
1	1	1

Logical Operations

The 6502 supports three kinds of logical operations: AND, OR, and exclusive-OR. All three operate on the accumulator using the contents of a memory location or, with immediate addressing, the contents of the second byte of the instruction.

An AND operation produces a 1 in each accumulator bit position in which both memory and the accumulator contain a 1; otherwise that bit position is reset to 0. See Table 3–1.

An OR operation produces a 1 in each accumulator bit position in which either memory or the accumulator (or both) contains a 1. See Table 3–2.

An exclusive-OR operation produces a 1 in each accumulator bit position in which either memory or the accumulator (but not both) contains a 1. See Table 3–3.

6502 Instructions Used for Logical Operations

The 6502 has three logical instructions:

Instruction	Description
AND	AND Memory with Accumulator
ORA	OR Memory with Accumulator
EOR	Exclusive-OR Memory with Accumulator

TABLE 3–2. Logical OR Operation

Memory Bit	Accumulator Bit	Result in Accumulator
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 3–3. Logical Exclusive-OR Operation

Memory Bit	Accumulator Bit	Result in Accumulator
0	0	0
0	1	1
1	0	1
1	1	0

EXERCISES

1. Write a program to subtract the 24-bit signed binary number in memory locations $00, $01, and $02 (most significant byte in $02) from the 24-bit signed binary number in memory locations $03, $04, and $05. Store the result in locations $03 through $05, replacing the minuend. Write the program code below. The structure of this program will be similar to that of the double-precision addition program you developed in step 3 of Chapter 2 (Chapter 2).

2. a. Run your program to perform the following subtraction:

   $267BFE–$74A017

   List the contents of the operands and result in the following table:

Subtrahend			Minuend			Results			
$00	$01	$02	$03	$04	$05	$03	$04	$05	Carry =

   b. What does the resulting state of the carry flag indicate for this particular case?

   c. Is the result valid or invalid? What did you check to find out whether or not the result is valid?

TABLE 3–4. Logic Information for System of Eight Devices

Bit Position	Device	Logic State Information
0	Pressure Sensor	1 = Pressure Above Setpoint 0 = Pressure At or Below Setpoint
1	Temperature Sensor	1 = Temperature Above Setpoint 0 = Temperature At or Below Setpoint
2	Velocity Sensor	1 = Velocity Above Setpoint 0 = Velocity At or Below Setpoint
3	Flow Rate Sensor	1 = Flow Rate Above Setpoint 0 = Flow Rate At or Below Setpoint
4	Concentration Sensor	1 = Concentration Above Setpoint 0 = Concentration At or Below Setpoint
5	Valve A	1 = Valve A Open 0 = Valve A Closed
6	Valve B	1 = Valve B Open 0 = Valve B Closed
7	Power	1 = Power On 0 = Power Off

3. a. Consider an industrial control system that has eight different devices, listed in Table 3–4. A byte in memory reflects the status of these devices at any given point in time. For example, a binary pattern of 11100010 in the status byte indicates that pressure is at or below the setpoint, temperature is above the setpoint, velocity is at or below the setpoint, and so on.

   Write a program to monitor the eight devices in the system, based on the relationship of two status bytes. The *current status byte* is stored at location $00 and the *prior status byte* is stored at location $01.

   Your program should do the following:

   a. Check to see which devices have changed state (exclusive-OR operation).
   b. Store the result of step a in location $02.
   c. Determine which devices have changed from logic 1 to logic 0 (AND operation).
   d. Store the result of step c in location $03.
   e. Determine which devices have changed from logic 0 to logic 1 (exclusive-OR operation).

f. Store the result of step e in location $04.

Location	Contents
$00	Current Status Byte
$01	Prior Status Byte
$02	Bits that changed state
$03	Bits that changed from 1 to 0
$04	Bits that changed from 0 to 1

Flowchart the program below.

b. Write the source code below.

TABLE 3–5. Summary of CSB and PSB

Device	PSB	CSB
Pressure Sensor		
Temperature Sensor		
Velocity Sensor		
Flow Rate Sensor		
Concentration Sensor		
Valve A		
Valve B		
Power		

4. a. Referring to the problem in step 3, assume that the current status byte has been found to be

$$CSB = 11101010_2$$

and the prior status byte is

$$PSB = 11101001_2$$

List the physical state of each device (above setpoint, etc.) in Table 3–5 for both the CSB and PSB.

b. Run your program for the preceding CSB and PSB conditions and list the results below.

Devices that changed:_____

Devices changed from 1 to 0:_____

Devices changed from 0 to 1:_____

CHAPTER FOUR

Program Sequencing

OBJECT

To perform loop and branch operations.

NEW INSTRUCTIONS

This chapter describes the branch instructions and the JMP, BIT, increment (INC, INX, and INY), decrement (DEC, DEX, and DEY), and compare (CMP, CPX, and CPY) instructions.

DISCUSSION

Computer programs are sequential by nature, in that they carry out a specified task by performing a series of instructions one after the other. It is important to realize, though, that the order in which the instructions appear on a coding sheet—and within the computer's memory—can be quite different from the *sequence* in which they are executed by the microprocessor.

The microprocessor, hereafter referred to as the CPU (for central processing unit), fetches program instructions from memory based on the address contained in the program counter (PC). At the beginning of a program, the program counter must contain the memory address of the program's first instruction. The CPU will fetch the instruction stored at that address, execute the instruction stored at that address, then update the program counter so that it "points to" the next instruction in memory.

None of the instructions in the preceding chapters (except JMP) altered the program counter; they simply allowed the PC to be updated to address the next consecutive instruction. However, the 6502 has two types of instructions that can change the contents of the program counter, and thereby alter the program sequence. These are:

- Branch instructions, which either change or do not change the program counter, depending on the state of a bit in the status register. If the test condition is satisfied, the CPU adds a signed *offset* to the program counter. This causes the next instruction to be fetched from a location that is some number of bytes higher or lower in memory than the branch instruction.
- Jump instructions, which load a new address into the program counter. Jump instructions are unconditional, that is, the jump is taken (the program counter is changed) whenever the jump instruction is executed. One additional jump instruction, JSR (jump to subroutine), will be covered in Chapter 8.

6502 Instructions for Branches and Jumps

The following branch and jump instructions apply to this exercise:

Instruction	Description
BCC	Branch on Carry Clear (C = 0)
BCS	Branch on Carry Set (C = 1)
BEQ	Branch on Result Zero (Z = 1)
BMI	Branch on Result Minus (N = 1)
BNE	Branch on Result Not Zero (Z = 0)
BPL	Branch on Result Plus (N = 0)
BVC	Branch on Overflow Clear (V = 0)
BVS	Branch on Overflow Set (V = 1)
JMP	Jump to New Location

The branch instructions operate only with relative addressing; the operand must be an 8-bit signed offset, relative to the second byte of the branch instruction. This means that the instruction to be branched to must be no more than 127 bytes past, or 128 bytes ahead of, the second byte of the branch instruction.

To save you the hassle of calculating the offset, most assemblers will accept an absolute address or (more typically) a *label,* and convert it to an 8-bit signed offset.

Therefore, your branch instructions will usually look like this:

```
Bcc  label
```

where "label" is a label on the "target" instruction.

For example, the instruction

```
        BCC THERE
HERE    ..
        ..
THERE   ..
        ..
```

branches to the instruction at the label THERE if the carry flag is clear (equal to zero) and continues to the instruction at label HERE if the carry flag is set (equal to 1).

The jump (JMP) instruction can operate with two different addressing modes, absolute and indirect absolute. Therefore, the operand can have these absolute forms:

```
JMP  label  or  JMP  addr
```

or these indirect absolute forms:

```
JMP  (label)  or  JMP  (addr)
```

The 6502 Compare Instructions

The 6502 provides instructions that permit you to compare the contents of a register (accumulator, X, or Y) with an immediate operand or a value in memory. The results of this comparison are recorded only in the status register; *neither operand is affected*.

The compare instructions are:

Instruction	Description
CMP	Compare Memory and Accumulator
CPX	Compare Memory and X Register
CPY	Compare Memory and Y Register

TABLE 4–1. Compare Instruction Results

Condition	N*	Z	C
A, X, or Y Less than Memory	1	0	0
A, X, or Y Equal to Memory	0	1	1
A, X, or Y Greater than Memory	0	0	1

*N is valid only for signed compares.

Essentially, these instructions subtract the contents of the memory byte from the specified register, and record the result of the operation in three flags in the status register:

- The carry (C) flag is set if the register is greater than or equal to memory, and reset if the register is less than memory.
- The zero (Z) flag is set if the contents of the register and memory are identical; otherwise it is reset.
- The negative (N) flag is meaningful only when signed numbers are being compared. N is set if the register is less than memory, and is reset if the register is greater than or equal to memory. Table 4–1 summarizes the compare flags.

Compare instructions are used to set up the N, Z, and C flags for subsequent testing by a branch instruction. For example, if you had a program that needed to take one of two paths, depending on whether the X register was equal to 100 or not equal to 100, the program might include this kind of sequence:

```
        CPX   #100
        BEQ   EQ100
NOT100  ..
        ..
EQ100   ..
        ..
```

Bit Test Instruction

The BIT instruction performs a logical AND between the addressed memory location and the accumulator, and has the following effects on the status register:

- The zero (Z) flag is set if memory and the accumulator have no "one" bits in common, and is reset otherwise.
- The overflow (V) flag receives the value of bit 6 of the addressed memory byte.
- The negative (N) flag receives the value of bit 7 of the addressed memory byte.

The BIT instruction is often used to test the status of some external device. For example, if the on/off state of a motor is reflected in bit 2 of a location labeled MOTOR, the following sequence could be used to wait for the motor to come on:

```
        LDA   #4        ;Mask off all bits except Bit 2
LOOP    BIT   MOTOR     ;Bit 2 = 1?
        BEQ   LOOP      ;Keep checking until it is
  ON    ..              ; then continue here
        ..
```

Instructions to Increment/Decrement Registers and Memory

The Compare and Branch instructions are often used in program loops, in which an operation is repeated a specified number of times. In such cases, the loop count is maintained in either a memory location or a register (X or Y). The 6502 instructions to increment or decrement memory or register contents are:

Instruction	Description
DEC	Decrement Memory by One
DEX	Decrement X Register by One
DEY	Decrement Y Register by One
INC	Increment Memory by One
INX	Increment X Register by One
INY	Increment Y Register by One

The increment and decrement instructions affect only two flags in the status register, zero (Z) and negative (N).

For example, if you wanted to repeat a certain operation five times, this loop would do the job:

```
        LDX   #5
LOOP    ..
        ..
        DEX
        BNE   LOOP
```

Normally the X and Y registers are used as counters, but if they are already in use in some other capacity, or if the counter must hold values greater than 255, you could use one or more *memory locations* to hold the count.

A Refresher on Absolute Indexed Addressing

In the upcoming exercise, you will be asked to find the largest of a *table* of numbers in memory. The easiest way to access elements in a table is to use one of the 6502's indexed addressing modes, either *zero page indexed addressing* (if the table is located in page zero) or *absolute indexed addressing* (if the table is elsewhere in memory).

As you may recall, the indexed addressing modes calculate an effective address by adding the contents of the X register or Y register to an address in the operand. The combinations are:

Mode	Operand Format	Effective Address
Zero Page Indexed, X	zpg-addr,X	EA = zpg-addr + X
Zero Page Indexed, Y	zpg-addr,Y	EA = zpg-addr + Y
Absolute Indexed, X	abs-addr,X	EA = abs-addr + X
Absolute Indexed, Y	abs-addr,Y	EA = abs-addr + Y

For example, if a table starts at location $11 and X contains the value 3, the instruction

```
LDA  $11,X
```

loads the contents of location $13 into the accumulator.

EXERCISES

1. Develop a program to arrange two single-byte, unsigned numbers in order of increasing value. The numbers are contained in locations $10 and $11. After the program has been executed, location $12 should reflect the outcome as follows:

   - ($12) = 0 if ($10) was less than ($11).
   - ($12) = 1 if ($10) was equal to ($11).
   - ($12) = 2 if ($10) was greater than ($11).

   Figure 4–1 is a flowchart of the required operations. Write the program in the space below. Use the X register for temporary storage if an exchange is required. As usual, the last instruction should be BRK.

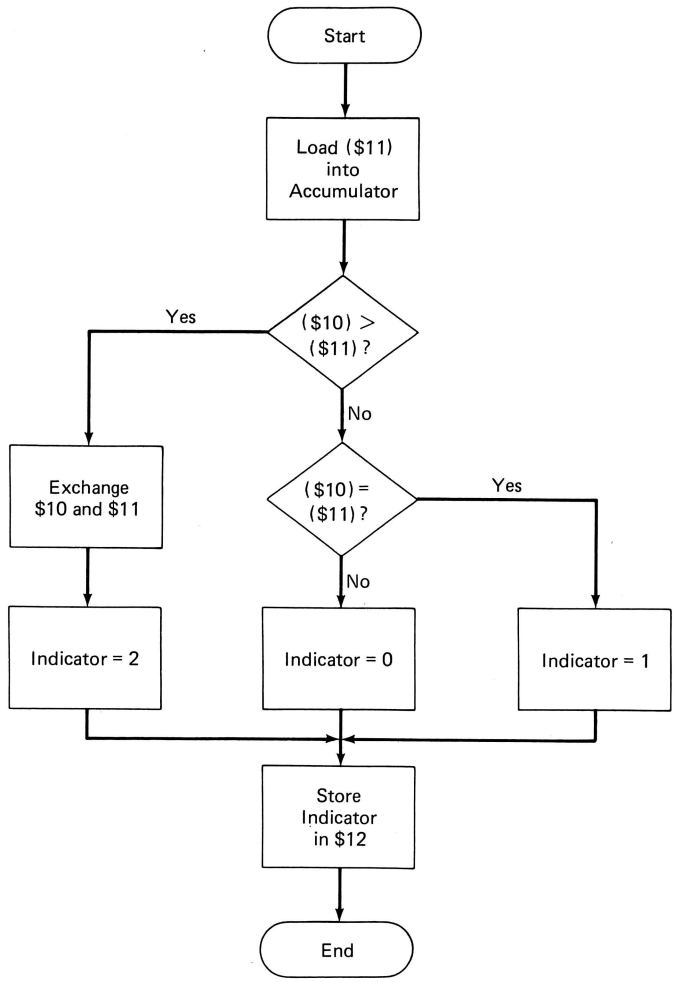


FIGURE 4–1. Flowchart to Arrange Two-Byte Values in Increasing Order

2. Run your program for the values given below and complete the $12 column for each case. You should also use the Monitor to check the final contents of locations $10 and $11.

$10		$11		$12
Dec.	Hex.	Dec.	Hex.	Hex.
150	96	14	0E	
150	96	150	96	
0	0	40	28	
0	0	0	0	

3. a. Find the largest unsigned number in memory locations $11 through $1A, and place this number in memory location $10. Flowchart the problem in the space below.

b. Write the program code below. Use indexed addressing to access the data table.

4. Run your program for the data given below and record the result in the space provided.

Location	Contents
$11	$A2
$12	$00
$13	$40
$14	$40
$15	$CE
$16	$5A
$17	$FA
$18	$55
$19	$06
$1A	$4F

Largest number ($10) = _____

CHAPTER FIVE

Debugging Programs

OBJECT

To identify errors, or "bugs," in programs.

PREPARATION

There are two different monitors for the Apple. The Apple II Plus has an Autostart Monitor, whereas all but the latest Apple IIs have a Standard Monitor. Both monitors have the commands described earlier, but the Standard Monitor has two additional commands, called *Step* and *Trace,* which can be useful for debugging programs. If your Apple has a Standard Monitor, read the debugging programs section of Chapter 3 in the *Apple II Reference Manual.*

DISCUSSION

Each of the preceding exercises involved drawing a flowchart of the program, writing the instruction sequence, then editing, assembling, and running the program. Unless all your programs operated correctly the first time, you had to do some debugging to remove the errors.

How can errors be identified? Until now, you probably tried a variety of approaches, everything from comparing the flowchart to the source code (always a good first step) to rewriting the program from scratch. These approaches will work for very short or very simple programs, but they become time consuming and inefficient for debugging more complex programs. The Apple II Standard Monitor includes two commands specifically designed as debugging aids. These are the Step and Trace commands.

The Step Command

The Step command causes one instruction in a program to be executed. Therefore, it can be used to execute a portion of a program, or an entire program, one instruction at a time.

When you execute the Step command, the Monitor disassembles and displays the instruction pointed to by the location counter, executes it, displays the contents of the microprocessor registers, and returns control to the Monitor.

The Step command has the format:

```
*aaaaS
```

where the address (aaaa) is optional. If you omit the address, the instruction at the current program counter address will be processed.

For example, if the instruction at location $300 is LDX #00, the command

```
*300S
```

will produce this display:

```
0300-   A2 00       LDX   #$00
 A=FF X=00 Y=8C P=32 S=F8
*
```

Since control has been returned to the Monitor, you can examine memory locations or do any other meaningful Monitor functions before resuming program execution.

The Trace Command

The Trace command displays the same disassembly and register information as the Step command, but executes instructions continuously until it encounters a BRK instruction or you press the RESET key. Your program will pay a time penalty for the trace service, however. Because the microprocessor must display information in addition to executing instructions, executing a program with the Trace command takes much more time than executing it with the normal Go command (*aaaaG).

The Trace command has the format:

```
*aaaaT
```

where the address (aaaa) is optional. If you omit the address, tracing will begin at the current program counter address.

As an example, consider a simple program that adds the contents of location $10 to location $11. The code for this program is

```
CLC
LDA10
ADC11
STA11
BRK
```

If the program is stored in memory beginning at location $300, the trace will look like this:

```
*300T

0300-   18           CLC
 A=FF X=00 Y=8C P=30 S=F8
0301-   A5 10        LDA   $10
 A=2E X=00 Y=8C P=30 S=F8
0303-   65 11        ADC   $11
 A=2F X=00 Y=8C P=30 S=F8
0305-   85 11        STA   $11
 A=2F X=00 Y=8C P=30 S=F8
0307-   00           BRK
0309-    A=2F X=00 Y=8C P=30 S=F8
*
```

How to Use the Trace Command

As a rule of thumb, to determine the execution path of a program (the sequence in which instructions are being executed), run the program under the Trace command and press RESET when you've seen enough to tell you what you want to know. This is very handy for determining whether certain instructions are ever getting executed, and for following the progress of a counter register.

To find out what some location in *memory* contains at a given point in your program, you can still use the Trace command, but before doing so it will be necessary to insert a BRK instruction into the program, at the instruction where the execution is to stop. For example, to determine the contents of location $11 prior to the STA $11 instruction, replace the first byte of the STA $11 instruction with a BRK instruction; that is, change the contents of location $305 from $85 to $00.

With this change made, the trace will look like this:

```
*300T

0300-   18           CLC
 A=FF X=00 Y=8C P=30 S=F8
0301-   A5 10        LDA   $10
 A=2E X=00 Y=8C P=30 S=F8
0303    65 11        ADC   $11
 A=2F X=00 Y=8C P=30 S=F8
0305-   00           BRK
0307-    A=2F X=00 Y=8C P=30 S=F8
*
```

And so, to find out what location $11 contains:

```
*11

0011- 01
*
```

Debugging with an Autostart Monitor

As mentioned earlier, the Autostart Monitor does not include the Step and Trace commands. Does this mean that readers who have this monitor and no debug program in their assembler package have no way of debugging a program? Of course not. These readers must simply insert a BRK instruction in their program whenever they need to investigate the contents of a register or a memory location. This gives the same results as the Trace command, but without the instruction-by-instruction display.

Where should BRK instructions be placed in a program? A few of the likely places include the following:

1. Following a loop, to investigate counter values and results in memory and registers
2. At instructions that serve as targets for branch instructions, to determine if and when the branch occurs
3. After a complex calculation, to check the result before continuing
4. After inputting data from a peripheral device, to see the data value that has been received

EXERCISES

1. In this exercise you will develop a program that counts the number of zeroes in memory locations $11 through $1A and places that count in location $10. Figures 5–1 and 5–2 show a flowchart and a symbolic program that should do the job. Enter the program and assemble it.

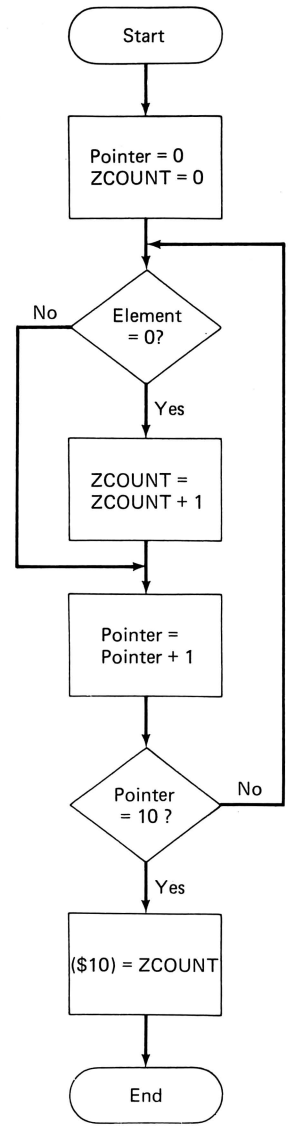


FIGURE 5–1. Flowchart for Zeroes-Counting Program

2. For now, assume that the program is error-free and see what results from a trial run. Initialize locations $10 through $1A with zero, then run the program.

3. Now check the result by displaying the contents of location $10, which contains the zeroes count. Since the ten-element data table (locations $11 through $1A) contained all zeroes, location $10 should have the value $0A, but instead you discover

   ($10) = 00

```
        ORG   $300
        LDX   #0        ;Index = start of table
LOADE   LDY   #0        ;Number of zeroes = 0
        LDA   $11,X     ;Fetch element
        BEQ   NEXT      ;This element = 0?
        INY             ;Yes.  Add 1 to zeroes count
NEXT    INX             ;Point to next element
        CPX   10        ;Any more elements?
        BNE   LOADE
        STY   $10       ;No.  Save zeroes count
        BRK             ; and return to Monitor
```

FIGURE 5–2. Source Code for Zeroes-Counting Program

What happened here? Since the BRK instruction was executed, you know that the preceding instruction, STY $10, was also executed, so the Y register definitely got stored into location $10. Since the value stored was *zero*, however, it appears that the zero-counting instruction, INY, never got executed. You'd better find out whether this was the case.

4. To find out whether INY was executed, disassemble the object code in memory (*300L) and find the address of the INY instruction. INY is stored at location $308, so replace INY with BRK by changing the contents of $308 from $C8 to $00.

5. Now run the program again. This should produce the display: of

```
030A-     A=FF X=0A Y=00 P=B5 S=DF
```

   Since location $030A is two bytes past the BRK instruction, you know that this break was caused by the BRK at $308, the former INY location, rather than by BRK at the end of the program. This tells you that the INY instruction *was* executed at some point in the program.

6. Where do you go from here? Well, you gain another item of useful information from the register display in step 5: you find out *when* INY was executed!
   If the program was working correctly, INY should have been executed when the first value from the table—the contents of location $11—was loaded into the accumulator. Since the instruction that fetches values from the table is

```
LDA   $11,X
```

the first value is fetched when X = 0. Note, however, the display shows X = $0A. That is, INY is being executed after the 6502 has fetched the contents of location $1B (that is, $11 + $0A).

7. Location $1B is beyond the range of the table, and should never have been referenced, but let's worry about one thing at a time. Namely, why was INY not executed when the value of location $11 was fetched?

   Under the ground rules, the zeroes counter (Y register) should have been incremented whenever a zero was loaded into the accumulator. However, look at the instruction that makes this "zero/nonzero" determination. It is

   ```
   BEQ  NEXT
   ```

   which means that if the accumulator contains zero, the 6502 is to *branch* to NEXT!
   However, the flowchart shows that this branch should take place if the accumulator does *not* contain zero. To remedy the problem, change the BEQ instruction to

   ```
   BNE  NEXT
   ```

   by replacing the BEQ's op-code ($F0) in location $306 with the op-code for BNE ($D0). Also restore the INY instruction, by changing location $308 from $00 to $C8. Verify these changes by disassembling the program again, then run the program.

8. What does location $10 contain now? A check should show

   ($10) = $00

   What can be wrong now? The Y register still doesn't appear to be incrementing?

> If your Apple II has a Standard Monitor, continue with Step 9. If your Apple II has an Autostart Monitor, skip to Step 9A, which follows Step 17 of the Standard Monitor procedure.

9. To find out what's going on, run the program with the Trace command. Figure 5–3 shows a portion of the display for the trace. The boxed lines indicate what's wrong with this program: After having been in-

```
*300T

0300-   A2 00       LDX   #$00
 A=00 X=00 Y=00 P=22 S=F8
0302-   A0 00       LDY   #$00
 A=00 X=00 Y=00 P=22 S=F8
0304-   B5 01       LDA   $11,X
 A=00 X=00 Y=00 P=22 S=F8
0306-   D0 01       BNE   $0309
 A=00 X=00 Y=00 P=22 S=F8
0308-   C8          INY
 A=00 X=00 Y=01 P=20 S=F8
0309-   E8          INX
 A=00 X=01 Y=01 P=20 S=F8
030A-   E4 0A       CPX   $0A
 A=00 X=01 Y=01 P=21 S=F8
030C-   D0 F4       BNE   $0302
 A=00 X=01 Y=01 P=21 S=F8
0302-   A0 00       LDY   #$00
 A=00 X=01 Y=00 P=23 S=F8
0304-   B5 01       LDA   $11,X
 A=00 X=01 Y=00 P=23 S=F8
0306-   D0 01       BNE   $0309
 A=00 X=01 Y=00 P=23 S=F8
0308-   C8          INY
 A=00 X=01 Y=01 P=21 S=F8
0309-   E8          INX
 A=00 X=02 Y=01 P=21 S=F8
```

FIGURE 5–3. Partial Trace of Zeroes-Counting Program

cremented to $01, the Y register is being reinitialized to $00, by the LDY #0 instruction at location $0302.

How did the microprocessor get back to *that* instruction? The instruction BNE $0302 at location $030C caused a branch to it.

You have just identified the second error in the program. This branch should cause the next element in the table to be fetched, but should not reinitialize the Y register, that is, the LOADE label should be on the instruction

```
LDA   $11,X
```

not on the LDY #0 instruction.

10. To make the branch instruction reference LDA, change location $030D (the offset) from $F4 to $F6, then run the program once more.

11. What does location $10 contain now? You should discover:

    ($10) = $01

    Apparently, there are still some problems with this program. Location $10 should contain $0A, since there are 10 zeroes between $11 and $1A. It appears the program has somehow processed just one element in the table.

12. To find out what is happening, run another Trace. The Trace shows that the program is making just one pass through the loop before returning to the Monitor. At this point it is reasonable to examine the instructions that *stop* the data-checking operation.

13. The terminating instructions in this program are CPX 10 and BNE LOADE, at locations $030A and $030C, respectively. The culprit is easy to identify: It is the CPX 10 instruction. This instruction is intended to compare the contents of the X register to the decimal value 10. Instead it is comparing the X register to the contents of *location* $0A (which has a decimal address of 10). The correct instruction is

    ```
    CPX  #10
    ```

14. To correct the instruction, change location $30A from $E4 (CPX with zero page addressing) to $E0 (CPX with immediate addressing).

15. Run the program one more time. At long last, the results are:

    (X) = $0A
    ($10) = $0A

16. To complete the job, use the editor's LOAD command to load the source code back into memory, then make the following corrections:

    - Move the LOADE label down one line, to the LDA instruction.
    - Change BEQ NEXT to BNE NEXT.
    - Change CPX 10 to CPX #10.

    The final version of the program is shown in Figure 5–4.

```
        ORG   $300
        LDX   #0
        LDY   #0
LOADE   LDA   $11,X
        BNE   NEXT
        INY
NEXT    INX
        CPX   #10
        BNC   LOADE
        STY   $10
        BRK
```

FIGURE 5–4. Final Version of Zeroes-Counting Program

17. Save the newly edited source code on disk (under a different file name), then assemble it and run it, to verify again that it is correct.

End of Standard Monitor Procedure

9A. To understand why the Y register was incremented only once, look at the instructions that terminate the loop. One is a compare instruction (CPX 10) and the other is a branch instruction (BNE LOADE). This means that looping should stop when the X register contains the value 10. But is this what those instructions do? No. The CPX instruction is comparing the X register to the contents of *location* $0A (decimal 10), rather than to the *value* 10! The instruction should read

CPX #10

To correct the instruction, change location $30A from $E4 (CPX with zero page addressing) to $E0 (CPX with immediate addressing), then run the program again.

10A. Now the display shows:

```
0312-    A=00 X=0A Y=01 P=33 S=E8
```

Finally the Y register has been incremented, but it has been incremented only *once!* And since location $10 holds the same value as Y, you will find

($10) = $01

However, you can see that the X register contains decimal 10 (hexadecimal 0A), so the program has tested ten locations.

11A. At this point, you can conclude that since INY is being executed, some instruction is *changing* the value in Y. Looking at the source code, you find only two instructions that alter the Y register: INY and LDY #0 at label LOADE. There's the answer! The LDY instruction has a label that's referenced by the loop-terminating instruction BNE LOADE. At the end of each loop, LDY *cleared* the Y register, thereby eradicating whatever value was in the register! The label should properly go on the next instruction:

```
LDA $11,X
```

The object code has no labels, but you can make this program change by altering the BNE instruction's second byte (the offset). Specifically, change the contents of location $030D from $F4 to $F6.

12A. Run the program once more. Now the display shows

```
0312-    A=00 X=0A Y=0A P=33 S=E6
```

and, at last,

($10) = $0A

The program is now bug-free!

13.A To complete the job, use the editor's LOAD command to load the source code back into memory and make the following corrections:

- Move the LOADE label down one line, to the LDA instruction.
- Change BEQ NEXT to BNE NEXT.
- Change CPX 10 to CPX #10.

The final version of the program is shown in Figure 5–4.

14.A Save the newly edited source code on disk (under a different file name), then assemble it and run it, to verify again that it is correct.

SUMMARY

Common programming errors for which to watch include:

1. Forgetting to initialize variables. Don't assume anything is zero, or any other value, when you start.
2. Confusing data and addresses. LDA 0 loads the contents of location 0 into the accumulator, whereas LDA #0 loads the value zero into the accumulator.
3. Branching on the wrong condition; for example, branching on "not equal" instead of "equal."
4. Confusing decimal and hexadecimal numbers. LDA #10 loads ten into the accumulator, whereas LDA #$10 loads sixteen into the accumulator.
5. Accidentally reinitializing a register or memory location by branching to the wrong place.
6. Making incorrect keyboard entries. List a source program on the display and examine it carefully before you assemble it.
7. Forgetting to update a counter or pointer. Watch for loop controls that must be updated regardless of which path the program takes.

CHAPTER SIX

Multiplication Operations with Shift and Rotate

OBJECT

To learn the shift and rotate instructions, and apply them to multiplication.

NEW INSTRUCTIONS

This chapter describes the shift instructions ASL and LSR and the rotate instructions ROL and ROR.

DISCUSSION

The 6502 has four instructions that cause the 8-bit contents of an operand (a memory location or the accumulator) to be displaced by one bit position to the left or to the right. Two of these instructions *shift* the operand, the other two *rotate* the operand.

For all four instructions, the processor status register's carry bit acts as a "ninth bit" extension of the operand. That is, carry receives the state of the bit that is displaced out of one end of the operand (bit 0 for a right shift, bit 7 for a left shift). In a *shift* operation, the vacated bit position at the opposite end of the operand (bit 7 for a right shift, bit 0 for a left shift) is reset to 0. In a *rotate* operation, the operand's vacated bit position receives the initial (prerotate) value of the carry bit.

The shift and rotate instructions are as follows:

Instruction	Description
ASL	Shift Left One Bit
LSR	Shift Right One Bit
ROL	Rotate Left One Bit
ROR	Rotate Right One Bit

The operations of these instructions are illustrated in Figure 6–1.

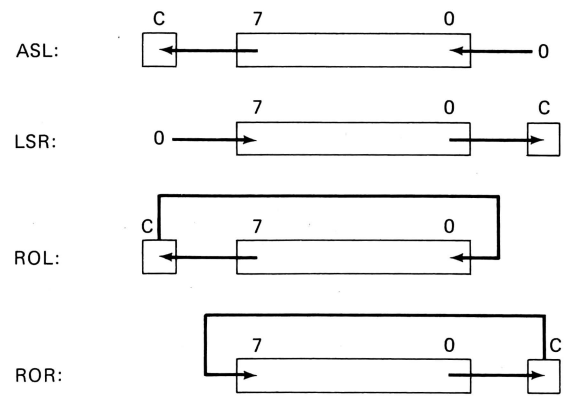


FIGURE 6–1. Shift and Rotate Instructions

As just mentioned, the shift and rotate instructions can operate on either the accumulator or a location in memory. If you specify A as the operand, the accumulator will be shifted or rotated. Hence,

```
ASL  A
```

causes the accumulator to be left-shifted.

Any other operand causes the contents of a memory location to be shifted or rotated. Thus,

```
ASL  $1000
```

causes the contents of location $1000 to be left-shifted.

The shift and rotate instructions are the only instructions that use the accumulator addressing mode.

In addition to their effect on the carry flag, the shift and rotate instructions affect two other flags in the processor status register:

- ASL, ROL, and ROR cause the negative (N) flag to be set if bit 7 of the shifted result is a logic 1; otherwise N is reset. The LSR instruction always resets the N flag, since it shifts a 0 into bit 7.
- The zero (Z) flag is set if the shifted result is zero; otherwise it is reset.

To show how these instructions work, consider an operand that contains a hexadecimal 34 (binary 00110100, decimal 52) and the carry flag is set to 1. Here is how the operand and carry flag would be altered for each of our four shift and rotate instructions:

Carry Flag	Bit Position 7 6 5 4 3 2 1 0	
1	0 0 1 1 0 1 0 0	Before Shift (= hex 34, dec 52)
0	0 1 1 0 1 0 0 0	After ASL (= hex 68, dec 104),
		or
0	0 0 0 1 1 0 1 0	After LSR (= hex 1A, dec 26),
		or
0	0 1 1 0 1 0 0 1	After ROL (= hex 69, dec 105),
		or
0	1 0 0 1 1 0 1 0	After ROR (= hex 9A, dec 154)

Multiplication Operations

You will be using the shift and rotate instructions to multiply binary numbers, but before you do, you should review the mechanics of decimal multiplication, as done with pencil and paper. As you know, you write the multiplicand on a piece of paper, with the multiplier below it, and perform a series of partial multiplications, one for each digit in the multiplier. Each partial product is written directly below its multiplier digit, causing this product to be displaced one digit position to the left of the preceding partial product. When all partial products have been calculated, they are added to produce the final product.

For example, the multiplication of 124 by 103 looks like this:

```
  124   Multiplicand
×103    Multiplicand
  372   Partial Product 1
  000   Partial Product 2
 124    Partial Product 3
12772   Final Product
```

In practice you normally do not write down the "all-zeroes" partial product, but it is shown here to emphasize one essential principle of multiplication: *A zero multiplier digit causes a skip to the next digit position without producing a partial product.*

Remember from third grade arithmetic that each partial product is displaced one digit to the left of its predecessor to reflect the higher decimal weight of its multiplier digit, that is, moving from right to left in the multiplier, each digit has a weight that is ten times greater than the digit to its right. For this reason, the preceding example could be written in this form:

$$103 \times 124 = (3 \times 124) + (0 \times 124) + (100 \times 124)$$

or, better yet, in this form:

$$103 \times 124 = (3 \times 10^0 \times 124) + (0 \times 10^1 \times 124) + (1 \times 10^2 \times 124)$$

With this groundwork, consider binary multiplication. Binary multiplication is much simpler than decimal multiplication because binary multipliers consist of only the digits 1 and 0. Therefore, the partial product is always a copy of the multiplicand (multiplier digit = 1) or zero (multiplier digit = 0).

The binary equivalent of our 103 × 124 example looks like this:

```
    01111100 Multiplicand (= 124)
  × 01100111 Multiplier (= 103)
    01111100
   01111100
  01111100
 00000000
00000000
 01111100
  01111100
00000000
011000111100100 Final Product (= 12772)
```

How can such a multiplication be performed with a computer? Well, like most 8-bit microprocessors, the 6502 has no multiply instruction, so the multiplication must be performed as we just did it: with a series of additions. However, some important differences exist between the pencil-and-paper approach and its computer counterpart. When multiplying by hand, you calculate all the partial products, then add them to generate the final product. In contrast, a computer program, instead of waiting until all partial products are derived before generating the final product, is much more efficient by updating the partial product after processing each multiplier bit. By maintaining this running total, the final product is available when the highest bit of the multiplier has been processed.

Just as the significance of digits in a decimal (or base 10) number increases by a factor of 10, the significance of digits in a binary (or base 2) number increases by a factor of 2, as you progress from right to left. So each binary digit is two times more significant than the digit on its right. As discussed previously, this increasing significance causes each partial product to be displaced to the left before writing it down on paper. With a computer program, it is easier to *rotate the current partial product one bit position to the right,* thereby aligning it to receive the next multiplier bit's contribution.

How long will the final product be? Well, our 103 × 124 binary multiplication had an 8-bit multiplicand and yielded a 15-bit final product. Since 2 bytes in memory will be needed to hold this 15-bit product, observe that a 1-byte multiplier and a 1-byte multiplicand yielded a 2-byte product. This was not coincidental; *the length of the product is always as long as the combined lengths of the multiplier and multiplicand.* (An n-byte multiplier and an m-byte multiplicand will produce an (n + m)-byte product.)

Figure 6–2 illustrates what you have learned about multiplication thus far by showing the sequence for processing one bit in the multiplier. Note that the state of

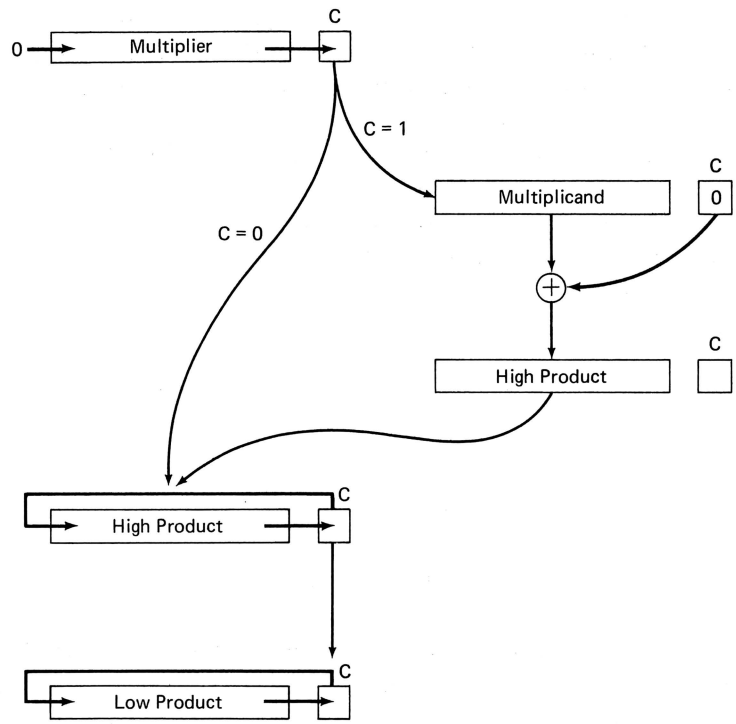


FIGURE 6–2. Processing a Multiplier Bit

the multiplier, shifted into carry, determines which of two possible paths are followed:

- If carry = 1, the multiplicand must be added to the high-order byte of the partial product before this product is rotated.
- If carry = 0, the partial product is rotated to the right, without an addition.

Programming Multiplication Operations

Multiplication programs should follow this general sequence:

1. Initialize the product to zero.
2. Shift the multiplier right, into carry, using the LSR instruction.
3. If carry = 1, add the multiplicand to the high-order byte of the partial product. If carry = 0, skip to step 4.
4. Rotate the partial product right, into carry, using two ROR instructions.
5. Repeat steps 2 through 4 for each bit in the multiplier.

To multiply multiprecision numbers, the most significant byte of the multiplier will be right-shifted with an LSR instruction. Less significant bytes must be right-rotated with ROR instructions.

EXERCISES

1. The discussion of shift and rotate instructions at the beginning of this exercise showed the effect of the ASL, LSR, ROL, and ROR instructions on the binary pattern 00110100, with the carry bit set to 1. Verify the results shown in the discussion by executing four 4-instruction programs of the form:

```
LDA  #$34
SEC
nnn
BRK
```

   where nnn differs between ASL, ROL, ROL, and ROR in the four programs.

2. a. Develop a program to multiply two 8-bit unsigned integers in memory. The multiplicand should be stored in location $11 and the multiplier should be stored in location $10. The 16-bit product should be returned in locations $12 (low byte) and $13 (high byte).
   Draw the flowchart below.

b. Write the program code below.

3. Run your program for the four cases listed below and complete the table.

Multiplier $10		Multiplicand $11		Product $12, $13	
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.
255	FF	1	01	255	
103	67	124	7C	12772	
255	FF	0	00	0	
255	FF	255	FF	65025	

CHAPTER SEVEN

Division Operations

OBJECT

To apply the shift and rotate instructions to division operations.

DISCUSSION

Division involves calculating the number of times one number (called the divisor) can be subtracted from another number (called the dividend). The integer value of the resulting number of subtractions is called the quotient, and the "leftover" fraction is called the remainder.

For example, dividing 1265 by 13 with pencil and paper would look like this:

```
   0097  ← Quotient
13 )1265
   -0
    12
   - 0
    126
   -117
      95
     -91
       4  ← Remainder
```

Although it is almost second nature by now, using this pencil-and-paper method you actually obtain the quotient digits by recalling the "division tables" hammered into you in elementary school. As with multiplication, this method cannot be applied to a computer solution, because the 6502—like most 8-bit microprocessors—has no divide instruction. In Chapter 6, multiplication was performed with a series of add operations. Division, similarly, can be performed with a series of subtraction operations.

Specifically, division can be performed with a series of *trial subtractions*, using the 6502's compare instructions covered in Chapter 4. In these trial subtractions, the

divisor is compared with the partial dividend, beginning with the left-most (most significant) digit of the dividend. With this approach, the logical steps for the preceding divide operation are as follows:

1. How many times can 13 be subtracted from 1? The answer is 0, so enter 0 in the quotient and bring down the next digit, 2, to form 12.
2. How many times can 13 be subtracted from 12? The answer is again 0, so enter another 0 in the quotient and bring down the next digit, 6, to form 126.
3. How many times can 13 be subtracted from 126? The answer is 9, so enter 9 in the quotient and subtract 117 (9 times 13) from 126.
4. The subtraction leaves a remainder of 9, so bring down the final digit of the dividend, 5, to form 95.
5. How many times can 13 be subtracted from 95? The answer is 7, so enter 7 in the quotient and subtract 91 (7 times 13) from 95.
6. All of the digits of the dividend have been tested, so the division is complete. The quotient is 97 (ignoring leading zeroes) and the remainder is 4.

Dividing binary numbers is similar to dividing decimal numbers, but is much easier. With binary numbers, your quotient digits will always be either 0 or 1. If the divisor can be subtracted from the partial dividend, the quotent digit will be a 1, otherwise it will be a 0.

The binary equivalent of our divide-1265-by-13 example is shown in Figure 7–1. As with decimal division, a new quotient digit (1 or 0 for the binary version) is always entered to the right of the previously entered quotient digits. In a computer program, this is accomplished by shifting the old partial quotient to the left and entering a new quotient digit into the vacated least significant bit position.

The computer implementation also requires the dividend to be left-shifted, to form a partial dividend to which the divisor is compared (and, if possible, subtracted). The fundamental operations for dividing binary numbers with a computer are as follows:

1. Shift the quotient (initially zero) left, to provide a (least significant) bit position for the next quotient digit.
2. Shift the dividend left, displacing its most significant digit into carry.
3. Rotate the partial dividend left, bringing in the new dividend digit from carry.
4. Compare the divisor to the partial dividend.
5. If the divisor is less than or equal to the partial dividend, subtract the divisor from the partial dividend, and enter a 1 in the quotient.
6. If any digits remain in the dividend, go back to step 1.

The preceding procedure can be shortened from six steps to five based on this observation: *The dividend and quotient can share the same space in memory.* How is this possible? The answer is simple: The dividend and quotient can share the same

```
          00001100001
      ┌──────────────
1101) 10011110001
     -0
      10
     - 0
      100
     -  0
      1001
     -   0
      10011
     - 1101
        1101
       -1101
           01
         - 0
           010
         -  0
           0100
         -   0
           01000
         -    0
           010000
         -  1101
              100
```

FIGURE 7–1. Binary Division

space in memory because at any given time the length of the quotient is equal to the number of digits that have been left-shifted out of the dividend. You never need more quotient digit positions than are available with the current dividend.

In the memory they share, the dividend will occupy the high-order bit positions and the quotient will occupy the remaining, low-order bit positions. So steps 1 and 2 or the six-step divide procedure can be combined into a single step:

(1,2) Shift the dividend/quotient left, displacing the most significant digit of the dividend into carry.

Figure 7–2 shows this step and the partial dividend rotate (step 3).

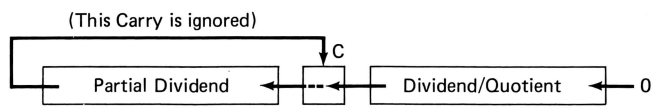


FIGURE 7–2. How Digits Are Entered into the Partial Dividend

EXERCISES

1. Develop a program to divide the 8-bit unsigned integer dividend in location $11 by the 8-bit unsigned divisor in location $10. The quotient and remainder, which are both 8-bit values, should be returned in locations $11 and $12, respectively.
   a. Draw the flowchart below.

b. Write the program code below.

2. Run your program for the four cases listed below and complete the table.

Divisor $10		Dividend $11		Quotient $11		Remainder $12	
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	Dec.	Hex.
1	01	255	FF	255		0	
15	0F	170	AA	11		5	
255	FF	1	01	0		255	
254	FE	255	FF	1		1	

3. As you know, dividing by zero produces an undefined quotient and is therefore illegal. What instructions can be added to the beginning of your program to check for a zero divisor, to bypass the divide operation if the divisor is zero? List the instructions below.

Add these instructions to your program, and run a test case to verify that they are correct.

CHAPTER EIGHT

Subroutines and the Stack

OBJECT

To program subroutines and use the stack.

NEW INSTRUCTIONS

This chapter describes the stack instructions PHA, PLA, PHP, and PLP, the subroutine instructions JSR and RTS, and the register transfer instructions TAX, TXA, TAY, TYA, TSX, and TXS.

DISCUSSION

In 6500-based systems, the memory locations in page 1 (addresses $0100 through $01FF) have a special purpose insofar as these locations are used as a *stack*. The stack functions as a temporary depository for register contents and for return addresses during execution of the break (BRK) instruction, a subroutine, or an interrupt (interrupts are discussed in Chapter 18).

Information is entered onto and extracted from the stack the same way we stack dishes in the kitchen. The last item to be placed on the stack is the first item to be removed from it. For this reason, the 6502's stack is usually referred to as "last-in, first-out." As register contents or return addresses are stored on the stack, they are deposited in read/write memory at lower and lower addresses; the stack "builds" toward address 0.

The 6502 contains an 8-bit register that maintains the address of the next "free" stack location. This register, the stack pointer, is automatically decremented by one after an information byte is *pushed* onto the stack, and is automatically incremented by one before an information byte is *pulled* off the stack. Because all stack addresses are of the form $01XX, the stack pointer holds only the low-order 8 bits of the address; the high-order 8 address bits ($01) are automatically added on by the 6502 at the time of a stack operation. For example, if the stack pointer contains $A3, the next byte of information will be pushed onto the stack at location $01A3 or pulled from the stack at location $01A4.

For all practical purposes, you can consider the stack's operation as being automatic, and you should not concern yourself with *which* stack location is being accessed at any given time. There are, however, two important rules for programming stack operations:

1. Every byte of information pushed onto the stack must eventually be pulled from the stack.
2. Information must be retrieved from the stack in the reverse order it was entered onto the stack.

Saving Register Contents on the Stack

The 6502 has instructions that permit the contents of both the accumulator and the processor status register to be saved on the stack. They are:

Instruction	Description
PHA	Push Accumulator on Stack
PLA	Pull Accumulator from Stack
PHP	Push Processor Status on Stack
PLP	Pull Processor Status from Stack

Aside from the fact that they push different registers onto the stack, the PHA and PHP instructions work identically. In each case, the contents of the register are stored at the location being pointed to by the stack pointer, then the stack pointer is decremented by one, to the next lower numbered address.

Similarly, the PLA and PLP instructions increment the stack pointer to the next higher numbered address, then load the contents of the addressed memory location into the appropriate destination register (A or P, respectively).

Figure 8–1 shows how a PHA instruction affects the stack. In the upper drawing the stack pointer (S) is pointing to the top of the stack, location $01FF, and the accumulator contains $03. After PHA is executed (lower drawing), the stack pointer has been decremented to $FE and the contents of the accumulator have been stored in location $01FF. If you execute a subsequent PLA instruction, the stack pointer would be incremented (to point to location $01FF) and the accumulator would be loaded with the contents of location $01FF.

Why would someone want to save register contents on the stack? For one reason only: to preserve their contents while the registers are being manipulated by other programming operations. This is particularly common in subroutines, but also applies to routines in which registers are serving two functions.

With only four stack instructions, it looks like only the accumulator and the processor status register can be saved on the stack. That is true as far as the instructions go, but the 6502's *register transfer instructions* allow you to move the X and Y registers into the accumulator, where they can be pushed onto the stack, later pulled

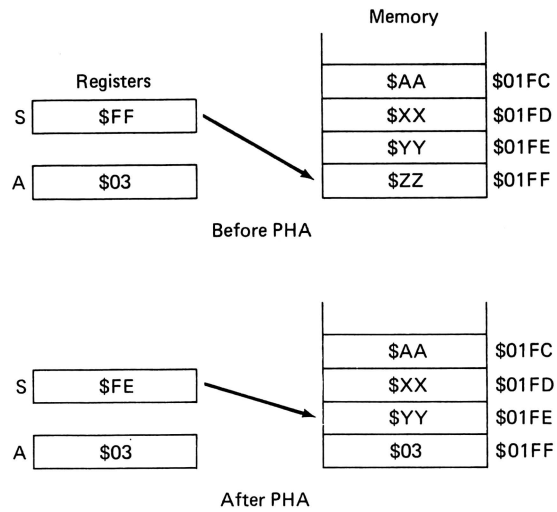


FIGURE 8–1. How a Register Is Pushed onto the Stack

off, then restored with additional register transfer instructions. The register transfer instructions are:

Instruction	Description
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

Figure 8–2 illustrates the transfer paths of the stack instructions and register transfer instructions, to help you remember them.

Subroutines

Many applications require a certain operation to be performed at several points in the program. To do this, you could rewrite the entire series of required instructions in each place they are used, but this would not only be frustrating and time consuming for the programmer, it would also make programs much longer than they would be if this repetition could be avoided.

As a matter of fact, all microprocessors do eliminate this need for recoding by permitting the repeated code to be defined as a *subroutine*. A subroutine is a sequence of instructions that is written just once, but can be executed as needed from any point in the program. The process of transferring control to a subroutine is given the term *calling*. Therefore, subroutines are *called*. Once called, the 6502 executes the sequ-

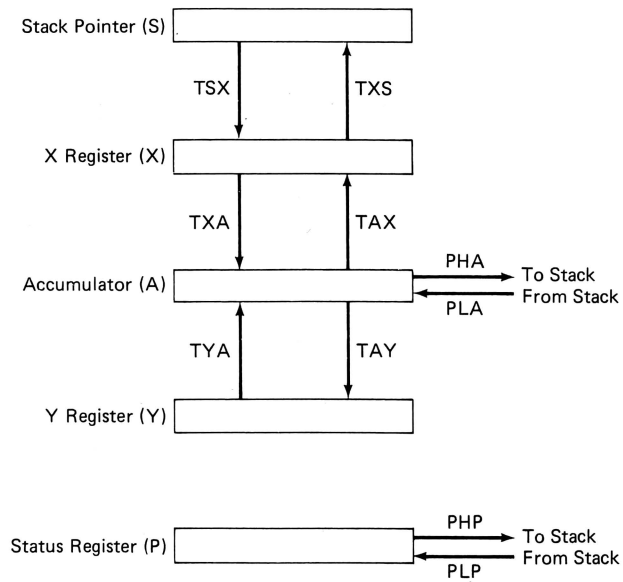


FIGURE 8–2. Register Transfer and Stack Instructions

ence of instructions in the subroutine, then returns control to the calling program.

From the preceding description, it is apparent that instructions which cause subroutines to be called must perform three functions:

1. They must include some provision for saving the contents of the program counter. Once the subroutine has been completed, this address will be used to return to the program.
2. They must cause the microprocessor to begin executing the subroutine.
3. They must use the stored contents of the program counter to return to the program, and continue executing the program at this point.

These three functions are performed by two 6502 instructions:

Instruction	Description
JSR	Jump to Subroutine
RTS	Return from Subroutine

The JSR instruction performs the return address-storing and begin-executing functions (1 and 2). The return address stored is the address of the third byte of the JSR instruction. (This address will be incremented by one, by the RTS instruction, to provide the proper destination.) The return address is stored on the stack, which means that the JSR instruction operates like the push instructions (PHA and PHP) described earlier in this chapter. Unlike PHA and PHP, which pushed 1 byte of data

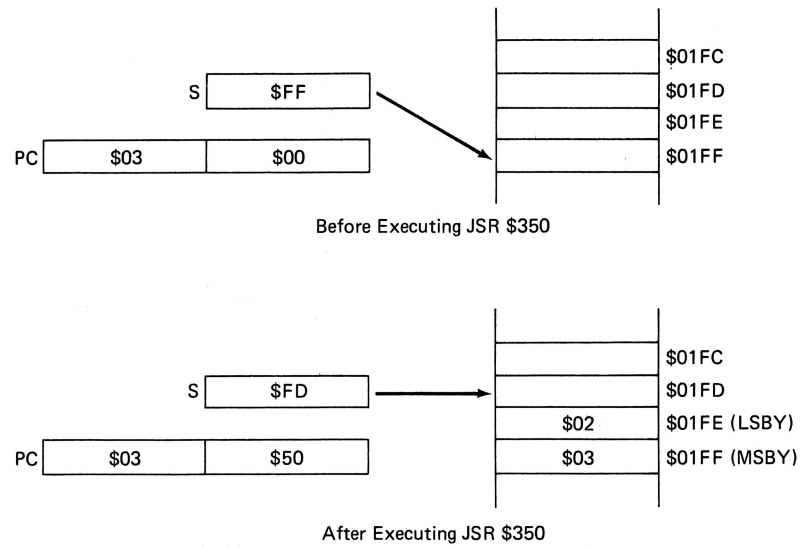


FIGURE 8–3. How JSR Saves a Return Address on the Stack

onto the stack, JSR pushes 2 bytes onto the stack—the 2-byte address in the program counter.

After storing the program counter contents on the stack, the JSR instruction loads the program counter with the absolute address from its second and third bytes, which transfers control to the starting address of the subroutine.

Consider a typical jump to subroutine instruction, JSR $350, located in memory locations $300, $301, and $302. Figure 8–3 shows the contents of the stack pointer (S), the program counter (PC), and the stack before and after the instruction JSR $350 is executed. In the upper drawing the program counter is pointing to the first byte of the JSR instruction ($300) and the stack pointer is pointing to the next free location on the stack ($1FF is assumed).

In the lower drawing, after the JSR $350 instruction has been executed, the program counter is pointing to the starting address, or *entry address*, of the subroutine ($350), the stack pointer is pointing to a new "next free stack location" ($1FD), and the top two bytes of the stack hold the address of the JSR instruction's third byte ($302).

The return from subroutine instruction (RTS) causes the 6502 to exit the subroutine and return to the calling program. In doing this, the RTS instruction causes the 6502 to continue program execution at an address one greater than the address on the top two bytes of the stack.

Recall that before loading the subroutine address into the program counter, the JSR instruction pushed the address of its third byte onto the stack. To retrieve that address, the RTS instruction increments the stack pointer, loads the low-order address byte into the lower half of the program counter, then increments the stack pointer again, and loads the high-order address byte into the upper half of the program counter. The program counter is then incremented so that it points to the instruction that follows JSR.

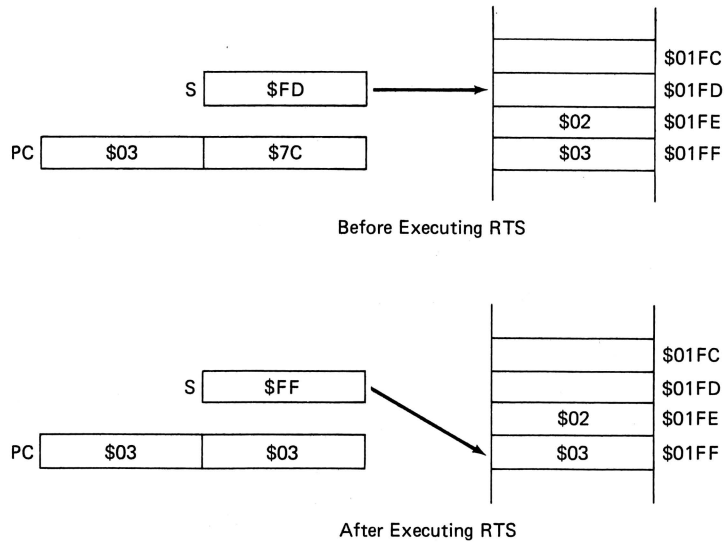


FIGURE 8–4. How RTS Pulls an Address from the Stack

Figure 8–3 illustrated the effects of the instruction JSR $350 on the stack. If the called subroutine contains an RTS instruction at location $037C, Figure 8–4 shows the contents of the stack pointer (S), the program counter (PC), and the stack before and after that RTS instruction is executed.

Subroutines Can Be Labeled

As with the jump instruction and the branch instructions, the operand field of a JSR instruction is nearly always a *label* rather than an absolute address. That is, in the source code, the JSR usually appears in the form

```
JSR  label
```

and the subroutine it calls has the corresponding form

```
label  ..
       ..
       RTS
```

Monitor Subroutines and the Equate (EQU) Directive

The Apple II monitor includes many useful subroutines your programs can call, which may save you hours of work spent "reinventing the wheel." Table 8–1 summarizes many of these built-in subroutines; others are described in Chapter 17.

TABLE 8–1. Some Built-in Monitor Subroutines

Entry Point	Name	Function	Registers to Load Before Calling	Registers Affected
$FDED	COUT	Output a Character to the Current Output Device.	A = Output Character Locs. $36 and $37 = Character Output Subroutine (Usually COUT1)	None
$FDF0	COUT1	Output a Character to the Screen's Text Window.	A = Output Character	None
$FC62	CR	Send a Carriage Return and Line Feed to the Screen.	None	A, X, and Y
$FD8E	CROUT	Send a Carriage Return to the Current Output Device.	None	A = $8D (CR)
$F940	PRNTYX	Print Contents of Y and X as a Four-Digit Hexadecimal Number.	Y = Most-Significant Byte X = Least-Significant Byte	A
$F941	PRNTAX	Print Contents of A and X as a Four-Digit Hexadecimal Number.	A = Most-Significant Byte X = Least-Significant Byte	A
$F944	PRNTX	Print Contents of X as a Two-Digit Hexadecimal Number.	X = Number	A
$FDDA	PRBYTE	Print Contents of Accumulator as a Two-Digit Hexadecimal Number.	A = Number	A
$FDE3	PRHEX	Print Lower Nybble of Accumulator as One Hexadecimal Digit.	A = Hex. Digit	A
$F948	PRBLNK	Print Three Spaces.	None	A = $A0 (SP) X = 0
$F94A	PRBL2	Print 1 to 256 Blank Spaces.	X = No. of Blanks (X = 0 prints 256 blanks.)	A = $A0 (SP) X = 0

$FF3A	BELL	Output a "Bell" Character to the Current Output Device.	None	A = $87 (BEL)
$FBDD	BELL1	Beep Apple II Speaker for 1/10 Second.	None	A and Y
$FD0C	RDKEY	Input a Character from the Current Input Device.	Locs. $38 and $39 = Character Input Subroutine (usually KEYIN)	A = Character Y
$FDIB	KEYIN	Read the Apple Keyboard.	None	A = Character
$FD6A	GETLN	Input a Line of Up to 256 Characters.	Prompt Character in Location $33 (usually *, for monitor.)	A and Y. X Contains Length of Data. Data Entered Starts at Loc. $200.
$FD67	GETLNZ	Same as GETLN, but Issues a Carriage Return Before Displaying the Prompt.	Same as GETLN.	Same as GETLN.
$FD6F	GETLN1	Same as GETLN, but Does Not Issue a Prompt.	None	Same as GETLN.
$FCA8	WAIT	Execute a Delay Loop Which Is $\frac{13 + 13.5x + 2.5x^2}{1.023}$ Microseconds Long.	A = Delay Value (x)	A = 0
$FF4A	SAVE	Save Register Contents in Reserved Page Zero Locations: Accumulator in $45 X Register in $46 Y Register in $47 Status Register in $48 Stack Pointer in $49	None	A and X
$FF3F	RESTORE	Restore Register Contents from Locations Used by SAVE.	None	A, X, Y, P, and S. See SAVE.

Note that Table 8–1 lists the Monitor subroutines by both starting address and name. If you're calling a subroutine just once in a program, it's easiest to call it with the starting address. For example, the instruction

```
JSR  $FDED
```

calls the Monitor's COUT subroutine.

However, if your program includes several calls to a Monitor subroutine, you can minimize the chance of mistyping the address, and make the source code more readable, by calling the subroutine by name. For example,

```
JSR  COUT
```

To call a subroutine by name, you must tell the assembler what address the name represents. This is done with a *equate* (EQU) *directive*. The equate directive has the general form

```
label  EQU  addr
```

where the label is always required; it is not optional. Equate directives should precede the source code instructions in a program, so a program that uses the COUT subroutine would look like this:

```
COUT  EQU  $FDED    (Equates start here)
      ..
      ..
      ORG  $300     (Main program starts here)
      ..
      ..
      JSR  COUT     (First call to COUT)
      ..
      JSR  COUT     (Second call to COUT)
      ..            (Remainder of program)
      ..
```

Equates are not restricted to subroutines. You can assign a label to *any* location you wish to reference by name.

Preserve Register Contents, If Necessary

The rightmost column of Table 8–1 shows the "registers affected" by each subroutine. These are registers whose original contents are *altered* during execution of the subroutine. If your program needs to use the original contents of those registers, they must be saved somewhere in memory before the subroutine is called. The easiest way to preserve register contents is to push these contents onto the stack before the

subroutine is called and pull them from the stack after the subroutine has been executed.

For example, the CROUT subroutine affects the accumulator. If you need to preserve the accumulator contents, the sequence of instructions may look like this:

```
..
..
PHA          (Save accumulator on stack)
JSR  CROUT   (Call CROUT)
PLA          (Restore the accumulator)
..
```

EXERCISES

1. Write a program to do the following tasks:
   A. Initialize the stack pointer to "point to" location $01FF.
   B. Load the accumulator, X register, and Y register with the values $01, $02, and $03, respectively.
   C. Push the contents of A, X, and Y onto the stack.

   Write the code for this program below. The final instruction should be BRK.

2. a. To see the effect of your program on the stack, examine the initial contents of locations $01FB through $01FF, and enter your findings in the BEFORE block below. After doing that, enter the program into the Apple II (starting at location $0300), execute it, then examine locations $01FB through $01FF again. Enter these new findings in the AFTER block below.

Before	Location	After
	$01FB	
	$01FC	
	$01FD	
	$01FE	
	$01FF	

b. If you added another PHA instruction at the end of this program, which memory location would receive the contents of the accumulator?

c. If instead of PHA you added a PLA instruction to the end of the program, which memory location would be accessed?

3. a. Since leaving values on the stack permanently is a "no-no," add some instructions to your program to do the following:
•Load A, X, and Y with the value $00.
•Pull the contents of A, X, and Y from the stack, then transfer these values into their former registers.
Write these additional instructions below.

b. Execute the new program. To verify the results, enter the final contents of A, X, and Y below.

Accumulator = X Register = Y Register =

4. a. To find out how subroutines operate, develop a subroutine that adds 5 to the contents of location $10. The subroutine should affect no registers; any registers used should have the same values upon return from the subroutine as they did when the subroutine was called. The subroutine instructions should start at location $300.
Write the code for the subroutine below.

b. Now write a program that initializes location $10 to zero, then *calls* your subroutine three times, which should yield a value of 15, or $0F, in location $10. The calling program should start at location $350.
Write the code for the calling program below.

5. Enter the calling program and the subroutine into the Apple II, then execute the program. Record the result below.

Location $10 =

CHAPTER NINE

Unordered Lists

OBJECT

To process unordered or randomly arranged data.

DISCUSSION

Information in memory can be organized for processing in many ways. These organizational techniques vary with the application and are categorized with such names as lists, arrays, strings, lookup tables, and vectors. As expected, the subject can (and does) fill many volumes, but this book concentrates on one of the most basic types of organization: *lists*.

Lists consist of units of data (one or more bytes), called elements, arranged sequentially in memory. The sequence can be *consecutive*, in which each element occupies one or more adjacent memory locations, or *linked*, in which each element includes an address "pointer" to the next element in the list. Moreover, elements can be arranged randomly, or in ascending or descending order. This chapter deals with unordered lists. The next chapter, Chapter 10, will discuss how to arrange the list's elements in order.

Unordered Lists

In our ordered society, where telephone book listings are arranged alphabetically and house numbers increase or decrease as you go up or down a street, unordered "anythings" seem somehow bothersome. However, unordered lists are a fact of life in many applications and represent a common way to store random data or data that change with time. For example, hourly temperatures over a 24-hour day or monthly shipments by a manufacturer would be stored in unordered lists.

Typically, lists are composed of an element count byte (or bytes) and one or more data elements. Three primary operations can be performed on a list: adding an element, deleting an element, and searching for a certain element value.

For a list whose elements are stored in consecutive memory locations, an element is added by storing it at the end of the list, then adding 1 to the element count. An element is deleted by simply moving the remainder of the list (all elements following

the one to be deleted) upward—toward address 0—by one element position, then subtracting 1 from the element count. Since the list is unordered, the search operation's procedure is apparent: Each element in the list, starting with the first element, must be compared with the search value. The search continues, element by element, until you find a match or reach the end of the list.

The compare portion of a search procedure follows the same rules outlined in Chapters 4 and 8: Single-byte numbers can be compared with one of the 6502's three compare instructions (CMP, CPX, or CPY), but multibyte numbers require a compare instruction to compare the least significant bytes and a subtract instruction (SBC) to compare all remaining, more significant bytes.

EXERCISES

1. In this exercise you will develop a program that searches an unordered list in memory, to find out whether the list contains the value being held in location $10. If the search value is found in the list, the number of the matching element (1, 2, 3, and so on) is returned in the Y register. If the search value is not found in the list, the Y register contains zero upon return.

   The starting address of the list is in locations $11 (low-address byte) and $12 (high-address byte). The length of the list, in bytes, is contained in the list's first byte (see Figure 9–1).

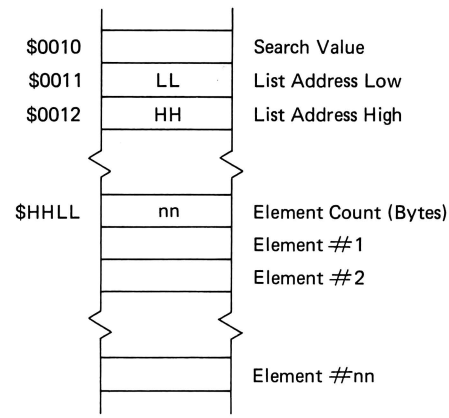


FIGURE 9–1. Unordered List in Memory

Flowchart the problem in the space below. The easiest solution involves using the X register to hold the element *count* from the list's first byte (decrement X as you advance through the list) and using the Y register to hold the *index* to elements in the list (increment Y as you advance through the list).

2. Write the program code below, then enter it into the Apple II, starting at location $300. Note that because the list's starting address is contained in page zero, your program must use indirect indexed addressing—with operands of the form ($11),Y—to access the list.

3. a. Assume that you wish to search for the value $55 in an unordered list that occupies locations $1000 through $100A. Store the following list values in memory:

Location	Contents
$1000	$0A
$1001	$A2
$1002	$00
$1003	$40
$1004	$40
$1005	$CE
$1006	$5A
$1007	$FA
$1008	$55
$1009	$06
$100A	$4F

Store the appropriate value in locations $10, $11, and $12, and record your entered values below.

Search value ($10) =

List address, low ($11) =

List address, high ($12) =

b. Run your program for a search value of $55 and record the result indication below.

Y Register =

4. The preceding program does nothing more than find out whether the search value is contained in the list. Now modify this program so that if it finds the search value, it will *delete* the matching element from the list.

   Draw a flowchart for the portion of the program that will delete the element if a match occurs. It should consist of a loop in which you decrement the count and move one element up in the list, until the count goes to zero (at which time you must decrease the count value in the list's first byte).

5. Write the code for a program that deletes an element from the list if a match occurs.

6. Using the list values from step 3, run your delete program for the search value $55, and record your answers below.

($1000) =

($1007) =

($1008) =

($1009) =

($100A) =

CHAPTER TEN

Sorting Unordered Data

OBJECT

To sort randomly arranged data.

DISCUSSION

Although many applications can handle data in unordered form (simply to plot or list items), it is often easier to process data that have been arranged in increasing or decreasing order. How can a list of unordered data be so arranged? One of the most common techniques is known as *bubble sorting.*

Bubble sorting arranges data elements in increasing order by causing higher valued elements to "rise" upward in memory (away from location 0), as soap bubbles rise upward into the sky. A bubble sort finds the relative magnitude of a data element by performing a series of comparisons. The list's first element is compared with the second element. If the first element is greater, the elements are exchanged. Then the second element is compared with the third element, exchanged if required, and so on. By the time the 6502 gets to the end of the list (completes one "pass" through the list), the largest valued element will have "bubbled up" to the list's last element position.

The bubble sort procedure requires the 6502 to make another pass through the list after each pass that included an element exchange operation. The exchange/no-exchange indication is usually provided by a special *exchange flag* in the program.

To see how the bubble sort actually operates, consider a five-element list that is initially arranged as follows:

05 03 04 01 02

After the first pass through the list, the arrangement will be

03 04 01 02 05

As you can see, the largest element in the list, 05, has bubbled up to the end of the list. The next pass will yield

03 01 02 04 05

Now the 04 value has bubbled up to the next-to-last position in the list. The next pass will complete the ordering operation, giving:

01 02 03 04 05

However, since this pass included an exchange (in fact, it included two exchanges), the microprocessor will make one more pass through the list. This final pass will produce no exchanges, so the sort is finished.

The preceding example not only demonstrates how bubble sorting works, but also indicates the performance you can expect from it. Note that four passes were required to sort a partially-ordered, five-element list. If the list was totally ordered at the outset, it would still take one pass for the 6502 to deduce this fact from the resulting status of the exchange flag. Conversely, if the list was initially arranged in descending order (the worst case), the bubble sort would require five passes to order the list.

The general rule is that the 6502 must make from one to N passes to bubble sort an N-element list. This means that an average of N/2 passes are required to sort an N-element list.

Processing Ordered Lists

How do the add, delete, and search operations on an *ordered* list differ from those same operations on an *unordered* list? The delete operation is identical for both types of list: an element can be deleted by moving all subsequent elements up (toward location 0) by one position, thereby overlaying the deleted element. As you may have guessed, adding an element to an ordered list involves a similar operation—move all higher-valued elements down (away from location 0) one position, insert the new element into the gap that has been created, and update the element count by one.

What about searching an ordered list? Can an ordered list be searched sequentially, element-by-element, as an unordered list searched? Of course it can, but a sequential search on an ordered list makes as little sense as searching for a number in a phone directory, name by name, from the beginning. Several well-known search techniques for ordered lists are faster and more efficient than the sequential search technique, for all but the shortest lists. Refer to any of the many books on programming techniques for more information on this subject. A few sources are:

1. Knuth, D. E. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley Publishing Co., Reading, MA., 1978. A classic treatment of the subject.

2. Scanlon, L. J. *6502 Software Design*. Howard W. Sams & Co., Inc., Indianapolis, IN, 1980, pp. 102–107. Describes binary search technique.
3. Bentley, J. L. "An Introduction to Algorithm Design," *Computer*, February 1979, pp. 66–78.

EXERCISES

1. a. Develop a program that uses the bubble sort algorithm to arrange the 8-bit elements of a list in ascending order. The starting address of the list is in locations $11 (low-address byte) and $12 (high-address byte). The length of the list is contained in the list's first byte.
Use location $13 to hold an *exchange flag*, an indicator that can be interrogated to determine whether any elements were exchanged during that pass (flag = $FF) or the pass was executed with no exchanges (flag = 0). Flowchart the problem in the space below.

b. Write the program code below, then enter it into the Apple II, starting at memory location $300. As in Chapter 9, you must use indirect indexed addressing—with operands of the form (11),Y—to access the list.

Hint: The most difficult part of this problem is writing instructions that exchange elements within memory. Since both X and Y are used in the program, they are unavailable for temporary storage during the exchange. As a solution, use the stack to hold the lower addressed element while you move the higher addressed element.

2. Store the following unordered list in memory:

Location	Contents
$1000	$0A
$1001	$A2
$1002	$00
$1003	$40
$1004	$40
$1005	$CE
$1006	$5A
$1007	$FA
$1008	$55
$1009	$06
$100A	$4F

Note that this is the same list used in step 3 of Chapter 9. The list has ten data elements (locations $1001 through $100A) and an element count byte (location $1000).
Order the preceding list using your program and record the result below.

Location	Contents
$1000	
$1001	
$1002	
$1003	
$1004	
$1005	
$1006	
$1007	
$1008	
$1009	
$100A	

CHAPTER ELEVEN

Code Conversion from Input

OBJECT

To convert input numbers to a form usable by the 6502 CPU.

DISCUSSION

All computer systems require a way of transfering digital information between the processor and external devices, that is, a way to *input* data from a tape, disk, or keyboard, as well as a way to *output* data to a printer, display, tape, or disk.

The data must, of course, be in a form recognizable to the recipient (the peripheral device or the microprocessor). Depending on its design, a peripheral device can operate on data in one of several forms, including ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), binary, or Gray code. The keyboard and display in an Apple II system operate with ASCII codes. When you press a key on the keyboard, an 8-bit ASCII code is placed on the system's data bus. Similarly, the display assumes that the information on the data bus is an ASCII code that represents the character to be displayed.

The 6502 microprocessor operates on numeric data in either of two forms, binary or BCD (binary coded decimal), depending on the state of the decimal mode (D) bit in the processor status register. The 6502 does not automatically convert these data into ASCII (or any other) form for output to a peripheral device, nor does it automatically convert ASCII characters from an input device into binary or BCD. How, then, is the information converted from one form to another?

Many computer systems have special electronic circuits that perform code conversions in hardware, but most general-purpose systems (such as the Apple II) require the conversions to be performed in software, with specially designed conversion subroutines. This chapter discusses subroutines that accept ASCII-based characters from the keyboard and convert them into binary data in memory. In Chapter 12 you will convert binary data in memory to one or more ASCII characters, and output these characters to the display screen.

Figure 11–1 summarizes the Apple's ASCII character set, showing the most significant digit (MSD) and the least significant digit (LSD) of the hexadecimal representa-

LSD \ MSD		8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	↑	n	~
F	1111	SI	US	/	?	O	←	o	DEL

NUL — Null
SOH — Start of Heading
STX — Start of Text
ETX — End of Text
EOT — End of Transmission
ENQ — Enquiry
ACK — Acknowledge
BEL — Bell
BS — Backspace
HT — Horizontal Tabulation
LF — Line Feed
VT — Vertical Tabulation
FF — Form Feed
CR — Carriage Return
SO — Shift Out
SI — Shift In

DLE — Data Link Escape
DC — Device Control
NAK — Negative Acknowledge
SYN — Synchronous Idle
ETB — End of Transmission Block
CAN — Cancel
EM — End of Medium
SUB — Substitute
ESC — Escape
FS — File Separator
GS — Group Separator
RS — Record Separator
US — Unit Separator
SP — Space (Blank)
DEL — Delete

FIGURE 11–1. ASCII Character Set for Apple II

tion for each character. For example, the letter B has a hexadecimal representation of C2 (MSD = C, LSD = 2).

Input Subroutine

Since the primary goal of Chapters 11 and 12 is to learn how to convert ASCII data to binary and vice versa, don't concern yourself with how the data are transferred between the 6502 microprocessor and external peripheral devices. For these tasks, you can use subroutines built into the Apple II Monitor.

To *input a keyboard character,* your program should call the Apple's KEYIN subroutine. This subroutine blinks the cursor until you press a key, then loads the

ASCII representation of that key into the accumulator. The KEYIN subroutine begins at memory location $FDIB, so your program should call it with the instruction:

```
JSR $FD1B
```

Types of Input Conversions

How many different kinds of input conversions must be accounted for? Well, since the 6502 microprocessor is designed to process both binary and BCD data, there must be subroutines that convert input data into these two forms. And since information is normally entered in either hexadecimal or decimal from an ASCII-based keyboard, three kinds of conversions need to be considered:

- ASCII-based hexadecimal to binary
- ASCII-based decimal to binary
- ASCII-based decimal to BCD

The fourth possible combination, ASCII-based hexadecimal to BCD, is so rare it is disregarded.

ASCII-Based Hexadecimal to Binary

Since a substantial amount of 6502 programming is done using hexadecimal data, many applications require that hexadecimal digits typed in at the keyboard be converted to binary codes. Table 11–1 shows the binary and ASCII equivalents for each of the 16 hexadecimal digits.

TABLE 11–1. The Hexadecimal Numbering System

Hexadecimal Digit	ASCII Value	Binary Value
0	B0	0000
1	B1	0001
2	B2	0010
3	B3	0011
4	B4	0100
5	B5	0101
6	B6	0110
7	B7	0111
8	B8	1000
9	B9	1001
A	C1	1010
B	C2	1011
C	C3	1100
D	C4	1101
E	C5	1110
F	C6	1111

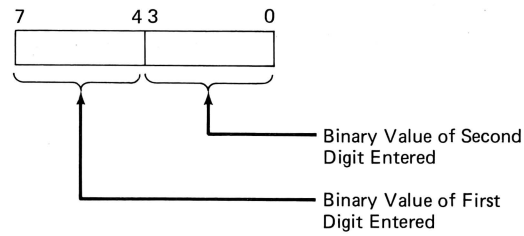


FIGURE 11–2. Two Converted ASCII Digits "Packed" into an 8-Bit Location

Table 11–1 shows what conversion is needed and gives a few ideas about how to do it. For starters, note that for the numeric values 0 through 9, the low-order hex digit of the ASCII value is identical to the hex character typed in. Obviously, for this range of values, the binary value can be obtained by simply stripping off the high-order hex digit of the ASCII value—which is a B. For the remaining values, A through F, if you strip off (or mask out) the ASCII value's high-order digit—a C—the remaining low-order digit is exactly 9 less than the required binary value. That is, $1 + $9 = $A, $2 + $9 = $B, and so on.

Most ASCII-based hex-to-binary conversion subroutines are designed to convert two ASCII-based hexadecimal characters to an 8-bit binary number. Why are *two* characters converted? Because each hexadecimal digit converts to a 4-bit binary value (0000 through 1111), and two 4-bit binary codes fit very nicely into an 8-bit memory location. Therefore, you convert two ASCII characters, and "pack" the two resulting 4-bit values into a single memory location. The high-order digit value is stored in the four high bits and the low-order digit value is stored in the four low bits, as shown in Figure 11–2.

ASCII-Based Decimal to Binary

In some respects, it is easier to convert ASCII-based decimal numbers to binary than to convert ASCII-based hexadecimal numbers to binary. However, as you shall see, the conversion process is more complex.

The ASCII-based decimal digits (0 through 9) are, of course, a subset of the ASCII-based hexadecimal digits (0 through 9, A through F), so the ASCII decimal/binary relationships are included in Table 11–1. As you can see from this table, if you are working with decimal digits, the only ASCII values of interest are those that fall between $B0 and $B9; all values less than $B0 and greater than $B9 do not represent decimal digits. You will also note that the binary equivalent of a decimal digit is nothing more than the four least significant bits of the ASCII character code.

An 8-bit register or memory location can hold the binary equivalent of decimal numbers between 0 and 255, so the basic ASCII-based decimal-to-binary conversion subroutine must be able to accept up to three ASCII characters and convert them to an 8-bit binary value (as long as the digits do not form a number larger than 255).

As you already know, decimal numbers can be expressed as a series of integers multiplied by powers of 10. For example:

$$237 = (7 \times 1) + (3 \times 10) + (2 \times 100)$$

or

$$237 = (7 \times 10^0) + (3 \times 10^1) + (2 \times 10^2)$$

Since digits of a number are entered one at a time, there will have to be a multiply-by-10 operation within any ASCII-based decimal-to-binary conversion subroutine. For instance, if 93 is entered, the 9 must be multiplied by 10 before being added to the 3.

Of course, a number can be multiplied by 10 using a multiplication routine such as that developed in Chapter 6. Alternatively, *a number can be multiplied by 10 by taking advantage of the fact that left-shifting a number doubles the value of that number.* This faster way to multiply a number by 10 is as follows:

- Left-shift the number twice, which multiplies it by 4.
- Add the original value to the result, which multiplies the number by 5.
- Left-shift that result once, which multiplies the number by 10.

With this background, you can define the required sequence of operations for a three-digit, ASCII-based decimal-to-binary conversion subroutine:

- For the first digit (most significant digit), the conversion subroutine must input the digit, convert it to binary, and store the binary value as a partial result.
- For the second and third digits, the conversion subroutine must input the digit, convert it to binary, multiply the stored partial result by 10, and add the new digit to the product (to update the partial result).

Since the Apple II Monitor accepts only hexadecimal operands, it has no built-in ASCII-based decimal-to-binary conversion subroutine that you can call. You will develop such a subroutine in the exercise at the end of this chapter.

ASCII-Based Decimal to BCD

Since the decimal digits (0 through 9) are a subset of the hexadecimal digits (0 through F), ASCII-based decimal-to-BCD conversion is similar to—but simpler than—the ASCII-based hexadecimal to binary conversion described earlier. For the BCD case, your program accepts only ten ASCII characters, consecutively numbered $B0 through $B9.

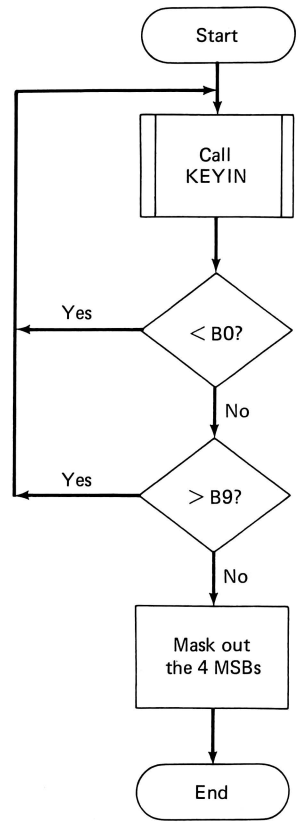


FIGURE 11–3. Keyboard Decimal Input Program

EXERCISES

1. You are to develop a program that inputs a decimal digit (0 through 9) into the accumulator and converts it from ASCII to a 4-bit binary value. Your program should use the Apple's KEYIN subroutine to input the keyboard digit (JSR FD1B) and should ignore any nondecimal character that is entered. Figure 11–3 shows the flowchart for this program. Write the code below.

2. Run your program three times, pressing the key(s) indicated below, and record the accumulator contents for each run.

Press Key(s)	Accumulator
9	
N, then 4	
2	

3. The program in steps 1 and 2 processes only a single digit from the keyboard. Now develop a program that accepts *three* decimal digits from the keyboard and converts them from ASCII to an 8-bit binary value in the accumulator. Since the accumulator is 8 bits wide, it can represent only decimal numbers between 000 and 255. If a number between 256 and 999 is entered, set the carry flag before return to signify an error condition.

   For this new problem, leave your single-digit conversion program in memory, but make it into a subroutine (by adding an RTS). The new, three-digit conversion program will call this subroutine three times, once for each digit. This problem will also require another subroutine, one that multiplies the partial result by 10 before it adds in the (converted)middle- and low-order digits.

   a. Draw the flowchart below.

b. Write the program code below.

4. Run your program for the four cases listed below and complete the table.

Entered Number	Result (Hex.)	Carry? (Yes/No)
000		
120		
255		
256		

CHAPTER TWELVE

Code Conversion for Output

OBJECT

To convert data to be output to external equipment.

DISCUSSION

Chapter 11 covered the operations a computer must perform to convert ASCII-based codes typed in at the keyboard to the binary and BCD forms that the 6502 can process. This chapter presents the operations that a computer must perform to convert binary and BCD information to ASCII-based characters recognizable to peripheral devices (including the Apple II system's display). Refer back to Figure 11–1 for a summary of the ASCII character set.

Output Subroutine

To output an ASCII character to the display, your program should call the Apple's COUT1 subroutine. Each time COUT1 is called, it places one character on the screen at the current cursor position (replacing whatever character was there) and moves the cursor one space to the right. The COUT1 subroutine starts at location $FDF0, so your program can call it with the instruction

```
JSR $FDF0
```

Types of Output Conversions

Because the 6502 can process two kinds of numeric data—binary and BCD—and these data may need to be displayed in either their hexadecimal or decimal representations, essentially three types of conversions must be considered:

- Binary to ASCII-based hexadecimal
- Binary to ASCII-based decimal
- BCD to ASCII-based decimal

The fourth combination, BCD to ASCII-based hexadecimal, is so rare that we disregard it.

Binary to ASCII-Based Hexadecimal

As you will recall from Chapter 11, two consecutive ASCII-based hexadecimal digits were converted into 4-bit binary codes and packed into a single 8-bit memory location. As shown in Figure 11–2, the most significant (first entered) digit was packed into the four high-order bit positions and the least significant (second entered) digit was packed into the four low-order bit positions.

Similarly, two packed 4-bit values can be converted to two ASCII-based hexadecimal characters by "unpacking" them. The usual procedure is as follows:

- Store the 8-bit value in memory (on the stack, if you like).
- Right-shift the accumulator four times to place the most significant digit's binary value in the four LSBs.
- Convert this value to ASCII-based hexadecimal by adding $B0 if the value is between 0 and 9 or adding $B7 if the value is between 10 and 15. See Table 11–1 if you are unsure as to why these values were used.
- Output the result as the most significant digit.
- Load the original value back into the accumulator.
- Mask out the four MSBs (using AND $0F).
- Convert this second value to ASCII-based hexadecimal by applying the preceding rules.
- Output the result as the least significant digit.

Figure 12–1 shows a flowchart for a subroutine that performs the preceding sequence of operations. Since each of the BCD digits must be converted to ASCII and output, this "main" subroutine calls another subroutine (also shown in Figure 12–1) which performs the conversion and outputs the result. Note that the second "call" to the conversion subroutine is not a *JSR*, but a *JMP*, which eliminates the need for an RTS instruction at the end of the main subroutine. Incidentally, the Apple Monitor's PRBYTE subroutine (entry address $FDDA) performs the same function as the subroutine in Figure 12–1.

Binary to ASCII-Based Decimal

Although computers process numbers in their binary or BCD form, it is often desirable to print results in decimal, to help decimally oriented humans understand them. Because this task is so common, it is worth discussing how binary numbers can be converted to their ASCII-based decimal equivalents.

Recall, from Chapter 11, that decimal numbers can be expressed as a series of integers multiplied by powers of 10. For example:

$$237 = (7 \times 10^0) + (3 \times 10^1) + (2 \times 10^2)$$

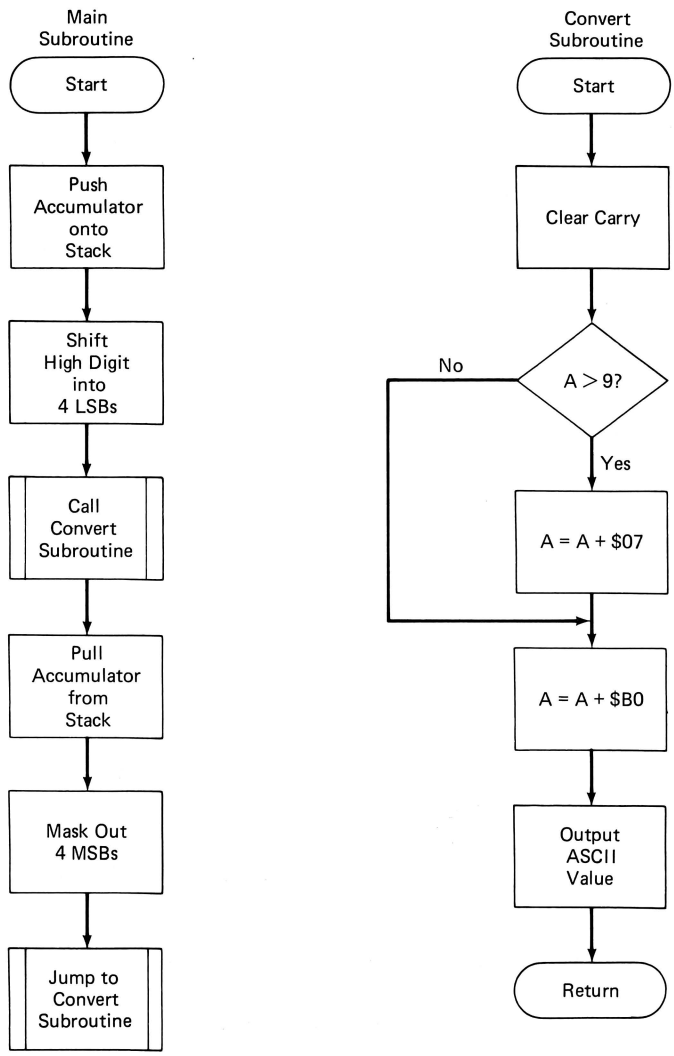


FIGURE 12–1. Binary to ASCII-Based Hexadecimal Subroutine

With this in mind, the primary job of converting an 8-bit binary number to a decimal number reduces to finding out how many 100's, 10's, and 1's are contained in the binary number. You can determine this by performing a series of successive subtractions on the binary value. The number of times that 100 can be subtracted from the binary value tells you the "hundreds" digit of the decimal representation. Similarly, the number of times that 10 can be subtracted from the remainder tells you the "tens" digit of the decimal representation, and the remainder of *that* operation tells you the "ones" digit.

For example, to determine the number of 100's in the decimal number 237, subtract 100 from 237 until another subtraction would produce a negative result. Each time

100 can be successfully subtracted, a *100's count* should be incremented by 1. For the number 237, the sequence is

```
 237
-100
 137   100's count = 1
-100
  37   100's count = 2
```

Now that the number of 100's is known, you can calculate a *10's count* with a similar subtraction sequence. Working with our previous remainder, 37, the sequence is

```
  37
 -10
  27   10's count = 1
 -10
  17   10's count = 2
 -10
   7   10's count = 3
```

At this point, you need not perform any more subtractions because the remainder, 7, represents the number of units (the *1's count*) in the answer.

The Apple II Monitor has no binary to ASCII-based decimal conversion subroutine. You will write such a subroutine as an exercise at the end of this chapter.

BCD to ASCII-Based Decimal

Since the decimal digits (0 through 9) are included in the set of hexadecimal digits (0 through 9, A through F), converting BCD values to ASCII-based decimal is similar to—but simpler than—converting binary values to ASCII-based hexadecimal. To convert BCD values, you would use the general subroutine shown in Figure 12–1, but its "convert" subroutine would *always* add $B0 to the accumulator, with no need to check whether the value in the accumulator (A) is greater than 9.

EXERCISES

1. a. Develop a program that converts an 8-bit binary value in the accumulator to a 3-digit, ASCII-based decimal string that is displayed on the screen. Use the Apple Monitor's COUT1 subroutine (JSR $FDF0) to perform the display operation.
   Draw the flowchart below.

   Hint: Use a compare instruction to determine whether the count value (100 or 10) can be subtracted from the remainder. If it can, perform the subtraction; if not, output the count and continue processing with the next decimal "weight."

b. Write the program code below.

2. Run your program for the four cases listed below and complete the table.

Number in Accumulator		Displayed Result
Dec.	Hex.	
030	1E	
255	FF	
002	02	
126	7E	

CHAPTER THIRTEEN

Timing Events with Software

OBJECT

To calculate time intervals with program instructions; specifically, to develop a program that calculates the time between two key depressions to the nearest one-tenth of a second.

PREPARATION

This exercise examines the operation of the keyboard and the design of the Monitor's KEYIN subroutine. Therefore, read the following parts of the *Apple II Reference Manual*:

- Reading the Keyboard, pages 6 through 8 in Chapter 1.
- Listing for the KEYIN subroutine in Appendix C. KEYIN occupies locations $FD1B through $FD2E.

The time interval between key depressions will be calculated using the Monitor's WAIT subroutine. Read its description in Chapter 3, page 63 of the *Apple II Reference Manual*.

You will also be using the binary to ASCII-based decimal conversion program developed in exercise step 1 of Chapter 12, so review that program if necessary.

DISCUSSION

Many applications involve time relationships of one kind or another. Some applications must measure the time between two events, such as the start and finish of a horse race or a marathon, or keep track of the elapsed time, as in a game program in which a player attempts to "beat the clock." Other applications require a specific time delay to be *generated*, rather than calculated, perhaps to turn on a drill for 10 seconds in a computer-controlled assembly line. You can probably think of several other time-related applications.

In this exercise, you will develop a program to do four things:

```
FD1B:  E6  4E      762  KEYIN   INC  RNDL     INCREMENT RANDOM NO. LOW
FD1D:  D0  02      763          BNE  KEYIN2
FD1F:  E6  4F      764          INC  RNDH      AND, IF NEEDED, HIGH
FD21:  2C  00 C0   765  KEYIN2  BIT  KBD      IS A KEY PRESSED?
FD24:  10  F5      766          BPL  KEYIN     NO. WAIT FOR ONE
FD26:  91  28      767          STA  (BASL),Y  YES. TURN OFF CURSOR
FD28:  AD  00 C0   768          LDA  KBD       READ ASCII KEY CODE
FD2B:  2C  10 C0   769          BIT  KBDSTRB   CLEAR KEY STROBE
FD2E:  60          770          RTS            AND RETURN
```

FIGURE 13–1. Monitor KEYIN Subroutine

1. Wait until a key (any noncontrol key) is pressed.
2. When a key has been detected, generate a 100-millisecond (1/10 second) time delay, then increment a counter register.
3. Wait for a second key stroke. It can be the same key as in step 1 or a different one.
4. When the second key has been detected, display the time interval as a three-digit decimal number, representing the elapsed time between key strokes, in tenths of a second.

Programming Keyboard Operations

In Chapter 1 of the *Apple II Reference Manual*, we learned that the keyboard circuitry "occupies" two locations in memory, $C000 and $C010. Location $C000 is a keyboard data location. It contains the ASCII character code for the last key (or key combination) that has been pressed. Location $C010 is a clear keyboard strobe location.

To find out how to interact with the keyboard, take a closer look at the construction of the Monitor's KEYIN subroutine. This subroutine is listed in Appendix C of the *Apple II Reference Manual* and is reproduced in Figure 13–1 for your convenience.

As you can see from the listing, KEYIN increments a 16-bit random number in zero page locations $4E and $4F, then waits until a key has been pressed. When the keystroke is detected, the microprocessor turns off the blinking cursor, reads the ASCII key code from the keyboard data register ($C000), then clears the key strobe signal ($C010).

This subroutine does much more than required for the time interval program in this exercise. However, you will be able to use the instructions that wait for a key to be pressed and the instruction that clears the key strobe. Keep these in mind for later reference. (*Comment*: Monitor listings can be very useful in developing programs because they show how "experts" approached a given task. These hints can often save you from spending hours of programming time "reinventing the wheel.")

A Time Interval Generator, the WAIT Subroutine

Now that you have dealt with the key detection tasks of your upcoming timing program, you can address the problem of timing the interval between the two keystrokes. Fortunately, the Monitor contains a subroutine to help you here, too.

This subroutine is called WAIT. Its entry address is $FCA8. The WAIT subroutine puts the microprocessor into a loop, in which it delays for a specific amount of time—based on the initial contents of the accumulator—and then returns to the calling program. WAIT affects neither the X register nor the Y register, but it clears the accumulator to zero.

The delay is given by the equation:

$$\text{Delay} = \tfrac{1}{2}(26 + 27A + 5A^2) \text{ cycles}$$

Since the Apple's 6502 microprocessor is driven at 1.023 MHz (that is, 1.023 million cycles/sec), the delay in cycles must be divided by 1.023 to derive the answer in microseconds. Therefore, the *correct* delay equation is:

$$\text{Delay} = \frac{13 + 13.5A + 2.5A^2}{1.023} \text{ microseconds}$$

Based on this equation, A = 1 generates a time delay of 28 microseconds and A = 255 generates a time delay of 162,285 microseconds. Due to the way the subroutine is designed, a value of A = 0 generates a delay of 163,547 microseconds.

The program to be developed here needs a 100,000-microsecond (or 1/10-second) delay. What value in the accumulator will cause a 100,000-microsecond delay to be generated? To find out, the delay equation must be solved for A. This can be done by applying the ever popular quadratic equation, which states that for equations of the form

$$aX^2 + bX + c = 0$$

the variable X has the solution

$$X = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In the delay equation, a = 2.5, b = 13.5, and c = 13 – 1.023D, where D is the delay in microseconds. Further, for the delay application, the minus (–) part of the plus-or-minus (±) can be disregarded. Therefore, the general equation solves to:

$$A = \frac{-13.5 + \sqrt{(13.5)^2 - 4(2.5)\ (13 - 1.023D)}}{5}$$

and then:

$$A = \frac{-13.5 + \sqrt{52.25 + 10.23D}}{5}$$

If you plug in D = 100,000 for a 1/10-second delay, you discover that A = 200 (decimal) for a 1/10-second delay.

Displaying the Time Count

All that remains now is to write a program to display the time between keystrokes. The time value will be generated as the contents of a single byte. To display this value as three decimal digits, use the conversion program developed in Chapter 12.

A cautionary note: If you have saved this program on tape or disk and load it into memory at a location other than the one at which it was recorded, you must modify the target addresses of JMP and local JSR instructions.

EXERCISES

1. a. Develop a program that calculates the time interval between two keystrokes, to the nearest 1/10 second. Assume that the time interval never exceeds 25.5 seconds, which allows the time count to be contained in a single byte.
   Draw the flowchart in the space below.

b. Write the program code below.

2. Test your program for 5, 10- and 15-second intervals between keystrokes using a wrist watch or wall clock. Record the results below.

Clock Interval	Displayed Interval
5.0 Seconds	
10.0 Seconds	
15.0 Seconds	

3. See how quickly you can press two keys, then see how quickly ou can press the same key twice.

Two keys: _____
_____
_____
_____ Average: _____

Same key twice: _____
_____
_____
_____ Average: _____

CHAPTER FOURTEEN

Apple II Sound

OBJECT

To program the Apple's built-in speaker to produce desired tones.

DISCUSSION

The Apple II contains a speaker that "occupies" location $C030. Each time the 6502 microprocessor references this location, with either a load or store instruction, the speaker emits a "click" sound. Admittedly, a single click doesn't sound too exciting, but if you make the speaker click fast enough, it sounds like a single tone or, in some ranges, like a musical note.

What is the highest pitched tone that can be produced? That tone will be produced by clicking the speaker as fast as possible, with a program like this:

```
ORG  $300
STA  $C030
JMP  $300
```

To save you the trouble of editing and assembling this short program, enter it in machine language, using the Monitor command

```
*300:8D 30 C0 4C 00 03
```

then execute it with the command *300G. What kind of tone do you hear being emitted? You should hear nothing, because this tone has a frequency of about *146,000 cycles/second,* and the upper limit of your hearing is around 20,000 cycles/second.

Now contrast this with the highest pitched tone a BASIC program can produce. Enter BASIC, then key in this program:

```
10 A = PEEK (-16336)
20 GOTO 10
```

and type RUN, followed by RETURN. Notice the difference? Here is an audible demonstration of the speed potential of assembly language versus BASIC. (In Integer BASIC, the highest frequency possible is about 256 cycles/second; in Applesoft BASIC, the highest frequency is about 72 cycles/second.)

Of course, the lowest possible tone will be produced by clicking the speaker just once, with the single instruction

```
STA   $C030
```

Try this with the Monitor command

```
300:8D 30 C0 00
```

In response to this instruction (actually two instructions, the second being BRK), the speaker does indeed click, but it's so brief that our ears—at least *my* ears—cannot hear it. Clearly, all meaningful tones lie somewhere between a single click and clicking with no delay, both of which are inaudible.

The tones generated by the Apple II's speaker are directly related to the *frequency* at which the speaker is clicked. The faster the speaker is clicked, the higher pitched the tone will be.

The hear this effect, you can use the Apple II Monitor's WAIT subroutine and experiment by changing the time between clicks (the *period*). Enter and run this program (omit the comments if you wish):

```
$300   LDA   #10      ;Accumulator determines the period
$302   JSR   $FCA8    ;WAIT for specified period
$305   STA   $C030    ;Click the speaker
$308   JMP   $300     ;Repeat the sequence
```

Since the program is terminated with a JMP $300 instruction, the only way to get out of it is by pressing RESET. Note that the accumulator had to be reloaded after each loop because WAIT always clears the accumulator. (Keep this in mind to avoid debugging problems later.)

Run the preceding program several times, changing the value in the accumulator each time. To do this, you needn't reenter the entire instruction; just change the contents of location $301 with the Monitor command

```
*301:hh
```

where hh is the two-digit hexadecimal new period.

After a while this manual operation begins to get on your nerves. Clearly, a much better way to change the delay value would be with a program that permits this value to be typed in from the keyboard. In the exercise that follows you will write such a program.

EXERCISES

You are to write a program that clicks the Apple II speaker with a frequency based on a delay (period) that has been typed in from the keyboard. The program should perform the following tasks:

- Read in a three-digit decimal number from the keyboard, and convert it to an 8-bit binary value in the accumulator. Use the program in Chapter 11 (Exercise step 3) to perform this task.
- Use the entered value as the delay input to the monitor WAIT subroutine.
- Click the speaker.
- Check for a new key from the keyboard. If a key has been pressed, return to step 1; otherwise, return to step 2 using the existing delay value as input to WAIT.

Remember, WAIT always destroys the contents of the accumulator!

1. a. In the space below, draw the flowchart for a program that reads the click period in from the keyboard.

b. Write the program code below.

2. Run your program for keyboard values of 001, 255, and several intermediate values. Observe the effects of these values on the tones that are emitted.

CHAPTER FIFTEEN

Let There Be Music!

OBJECT

To apply the sound-generating principles of Chapter 14 to produce musical notes on the Apple II speaker.

DISCUSSION

In Chapter 14 you learned through experimentation what effects various time delays had on the tones emitted through the Apple's speaker. However, this was done rather haphazardly. Unless you rigorously analyzed the tones, they remained more or less unrelated.

You probably guessed, though, that if the delay times could somehow be *correlated* with the tones produced, you would have the groundwork to build music programs for the Apple. Well, within the music field, well-defined relationships *do* exist between frequencies and musical notes. Piano tuners and tuners of other instruments use these relationships to tune their instruments.

Figure 15–1 shows a portion of a piano keyboard, spanning two octaves,* from low C to high C. In this drawing each key is labeled with its associated note name. The white keys represent the *natural notes*, labeled A through G. The black keys represent sharped and flatted notes, which produce tones one-half note higher (sharp) or one-half note lower (flat) than the white keys beside them.

Figure 15–1 also shows the *frequency* of each note, in cycles/second. With these data, it is possible to calculate a period (delay time) for each note. Further, since the Apple II keyboard has keys labeled A through G, it should be possible to design a music program that emits a "note" corresponding to each letter key pressed on the keyboard.

Table 15–1 shows the data you need to get started on a music program. It lists the notes and frequencies shown in Figure 15–1, along with the periods required between speaker clicks and the accumulator values needed by the Monitor's WAIT subroutine.

*Octave = eight notes. Here, the lower octave runs from low C to middle C and the upper octave runs from middle C to high C.

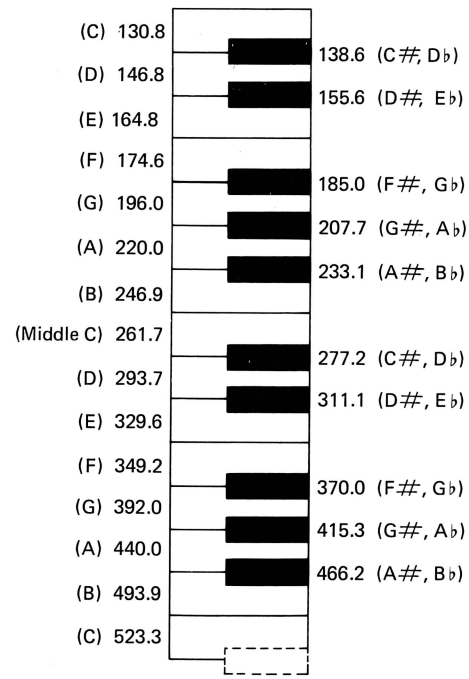


FIGURE 15–1. Portion of Piano Keyboard to Be Simulated

EXERCISES

You are to develop a program that reads in a key from the keyboard, then causes the speaker to produce the associated note continuously, until another key is pressed. For this particular exercise, limit yourself to the natural notes (A through G).

Since Table 15–1 includes two occurrences of each note (and *three* occurrences of C), the program must be able to distinguish between them. That is, it must be able to "decide" whether a G from the keyboard is intended to select the lower G (frequency = 196 cycles/second) or the higher G (frequency = 392 cycles/second). In the program you develop, *notes between low C and B should be selected when the corresponding letter key is pressed; notes between middle C and high B should be selected when both the letter key and CTRL are pressed.* (High C is a special case; you will see how to select it in Chapter 16.) For example, the G key (alone) should produce the lower G tone, whereas CTRL-G should produce the upper G tone. Hereafter, the notes between low C and B will be called the lower octave and notes between middle C and high C will be called the upper octave.

Use of the CTRL key gives your program sufficient information to make the low-octave/high-octave decision because it causes a different ASCII code to be generated. Table 15–2 shows the ASCII codes for the keys A through G, with and without the CTRL key.

In effect, then, your program must determine whether the selected note lies in the lower octave or the upper octave. Once this has been determined, the ASCII value

TABLE 15–1. Delay Values for Notes

Note	Frequency (Cycles/Sec)	Period (Sec)	Value in A Dec.	Value in A Hex.	
C	130.8	0.007645	53	35	
C#, Db	138.6	0.007215	52	34	
D	146.8	0.006812	50	32	
D#, Eb	155.6	0.006427	49	31	
E	164.8	0.006068	47	2F	
F	174.6	0.005727	46	2E	
F#, Gb	185.0	0.005405	44	2C	
G	196.0	0.005102	43	2B	
G#, Ab	207.7	0.004814	42	2A	
A	220.0	0.004545	40	28	
A#, Bb	233.1	0.004290	39	27	
B	246.9	0.004050	38	26	
C	261.7	0.003821	37	25	(Middle C)
C#, Db	277.2	0.003608	36	24	
D	293.7	0.003405	35	23	
D#, Eb	311.1	0.003214	34	22	
E	329.6	0.003034	33	21	
F	349.2	0.002864	32	20	
F#, Gb	370.0	0.002703	31	1F	
G	392.0	0.002551	30	1E	
G#, Ab	415.3	0.002408	29	1D	
A	440.0	0.002273	28	1C	
A#, Bb	466.2	0.002145	27	1B	
B	493.9	0.002025	26	1A	
C	523.3	0.001911	25	19	

must be converted into an *index* into a *look up table;* in this case a table of fourteen delay values, seven for the lower octave and seven for the upper octave.

You can use the assembler's define byte (DFB) directive to construct the lookup table. This directive has the general form

```
DFB byte, byte, byte . . .
```

and may be labeled. DFB directs the assembler to store the listed byte values into consecutive locations in memory. DFBs should be placed immediately *after* your program; if you put them *in* the program, the 6502 will attempt to execute the bytes as instructions.

If you call the delay table DLYTBL, the proper DFBs are

```
DLYTBL   DFB  $28,$26,$35,$32,$2F,$2E,$2B   ;Lower octave
         DFB  $1C,$1A,$25,$23,$21,$20,$1E   ;Upper octave
```

To access any particular element in the table, you will need an instruction of the form

```
LDA DLYTBL-1,X
```

where X, the index, holds a value from 1 to 14 ($01 to $0E).

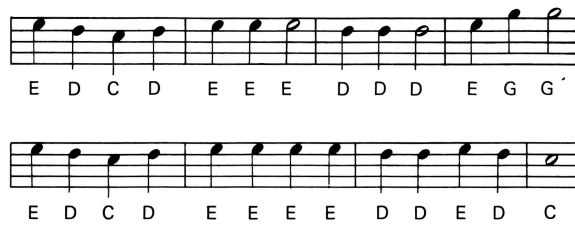
TABLE 15–2. ASCII Codes for Keys A through G

Key	Alone	With CTRL
A	$C1	$81
B	$C2	$82
C	$C3	$83
D	$C4	$84
E	$C5	$85
F	$C6	$86
G	$C7	$87

1. a. In the space below, draw the flowchart for a program that reads in a key, then causes the speaker to emit the note corresponding to that key continuously, until the next key is pressed.

b. Write the source code for the program in the space below.

2. Run your program to verify that the key sequence C, D, E, F, G, A, B, CTRL-C produces the familiar "do-re-mi-fa-so-la-ti-do" scale you learned as a child.
   Here are the notes for a familiar tune. All are in the upper octave, so hold the CTRL key down.



   What is the tune?_____

3. Some of the tones sound slightly "off-key." Give at least one reason why this is so, and propose a solution to the problem.

CHAPTER SIXTEEN

More Music

OBJECT

To generate the tones for sharped and flatted notes on the Apple II speaker.

DISCUSSION

In Chapter 15, you developed a program that allowed you to "play" the natural notes of a range of keys (Figure 15–1) on the Apple II keyboard. That program provides the base from which to expand the music-producing capabilities to include the sharped and flatted notes, the black keys on a piano.

As mentioned earlier, the sharped and flatted notes are one-half note higher (sharp) or lower (flat) than their corresponding natural notes. For example, D-sharp (D♯) is one-half note higher than D and D-flat (D♭) is one-half note lower than D. You will note from Figure 15–1 that there are no black keys between E and F nor between B and C. Therefore, E♯ is F, F♭ is E, B♯ is C, and C♭ is B.

Table 15–1 listed delay value inputs to the WAIT subroutine for each key shown in Figure 15–1. Table 16–1 gives this same kind of summary information in tabular form. You will recognize the "naturals" rows as the lookup table in your program for the Chapter 15 exercise. The sharps and flats rows give corresponding data for the sharped and flatted notes.

Notice that the values for E♯ are identical to those for F-natural, per the previous ground rules; both E♯ and F have values of $2E in the lower octave and $20 in the upper octave. This also holds true for values of F♭ and E-natural. But wait a minute, the value for B♯ lower matches the value for C upper—as it should, since we're changing octaves—but the value for B♯ upper ($19) is nowhere in the source table, Table 15–1! What has happened is that a new note, C above middle C, has been added to our "piano keyboard." That is, realizing that B♯ is really C, we are allowing you to select a new C-natural note, by typing B♯ in the upper octave (CTRL held down).

This new note, C above middle C, has a frequency of 523.3 cycles/second, which converts to a delay value of 25 decimal or $19 hexadecimal. Similarly, a new note can be added to the low end of the key range. This note, low B, is selected by typing C♭ in the lower octave (no CTRL). Low B has a frequency of 123.5 cycles/second, which converts to a delay value of 55 decimal or $37 hexadecimal.

TABLE 16–1. Delay Values for All Notes

	A	B	C	D	E	F	G	Octave
Naturals	28	26	35	32	2F	2E	2B	Lower
	1C	1A	25	23	21	20	1E	Upper
Sharps	27	25	34	31	2E	2C	2A	Lower
	1B	19	24	22	20	1F	1D	Upper
Flats	2A	27	37	34	31	2F	2C	Lower
	1D	1B	26	24	22	21	1F	Upper

Approach to Programming

The data in Table 16–1 give you a hint on how to develop a program to handle the sharps and flats, as well as the natural notes. This table can be viewed as three separate lookup tables: the table of natural notes (same as in Chapter 15), the table of sharps, and the table of flats.

In essence, your program must determine whether a natural note, a sharped note, a flatted note, or none of the three has been typed in at the keyboard. A nonnote should, of course, cause a return to the KEYIN subroutine. A valid note, however, should cause the program to look up the appropriate delay value and use this value as an input to the WAIT subroutine. As before, the selected note should sound on the speaker until a new key has been pressed.

The natural inclination is to write a program that will accept a two-key sequence for a sharp or flat in the order letter-then-♯ or letter-then-♭ (such as A♯ or A♭). Unfortunately, in a program it becomes nearly impossible to determine whether an A typed in represents an A-natural or just the first key in an A-then-♯ sequence. For this reason, you should design your program to accept the sharp or flat symbol *before* the letter it references, that is, the correct form should be ♯ A, to represent A♯!

The ♯ symbol on the Apple keyboard is part of the 3 key. However, the flat symbol, ♭, is not on the keyboard, so instead we'll arbitrarily use the 4 key. For ease of use, neither case should require SHIFT to be pressed. Therefore, to select A♯ in the lower octave, press 3, then A. To select A♯ in the upper octave, press CTRL-3, then CTRL-A.

EXERCISES

1. a. In the space below, draw the flowchart for a program that processes all music keys, including sharps and flats.

b. Write the source code for this program below.

2. Test your program by keying in all combinations of notes, sharps, flats, and naturals. Be sure to include the two new notes, C♭ (which gives low B) and CTRL-B♯, (which gives high C). Of course, also feel free to play any tunes you like.

CHAPTER SEVENTEEN

Low-Resolution Graphics

OBJECT

To program low-resolution graphic shapes on the screen.

PREPARATION

In Chapter 1 of the *Apple II Reference Manual,* read the sections titled "Screen Format," "Screen Pages," "Screen Switches," and "The Low-Resolution Graphics (Lo-Res) Mode."

DISCUSSION

Besides text, the Apple II can also display graphics with either low or high resolution. Programming high-resolution graphics in assembly language is a fairly difficult task and beyond the scope of this book. However, programming low-resolution graphics is relatively easy. The results can be satisfying too, making it worth your while to learn.

Low-Resolution Screen Pages

The Apple II has two areas of memory set aside to hold low-resolution graphics information. These areas, called *page 1* and *page 2,* are each 1024 bytes long. Only one of these pages can be displayed at any given time.

When the Apple II is operating in the graphics mode, the active page appears on the screen as a display of rectangles (or *graphic points)* 40 columns wide and 48 rows high. Figure 17–1 shows this display, with numbers assigned to the columns and rows. You will use these numbers in programming low-resolution graphics shapes.

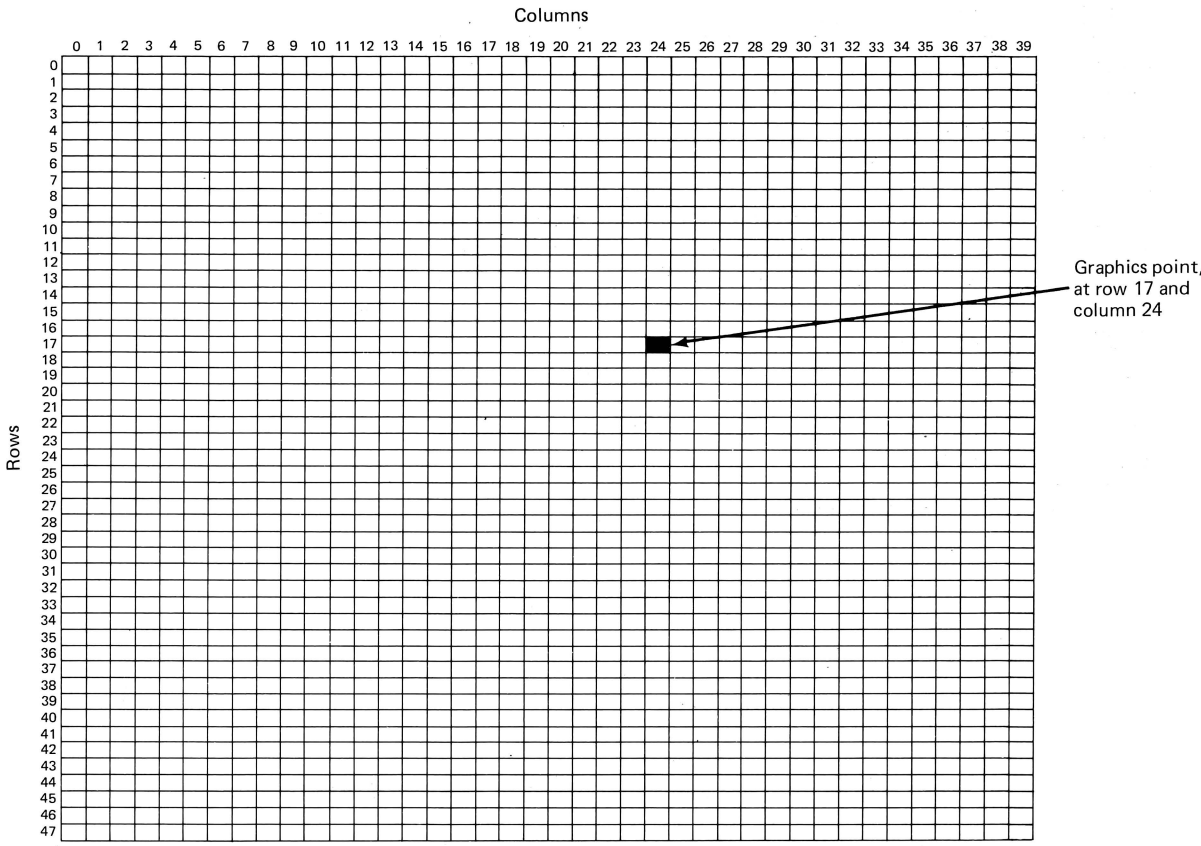


FIGURE 17–1. The Low-Resolution Graphics Screen

How to Select the Low-Resolution Graphics Mode

The Apple II's display modes are controlled by a set of memory locations called *soft switches*. Like light switches, each soft switch has two positions. They are called soft switches because they can be changed with software. A soft switch is changed by accessing one of its two locations with either a load or store instruction, just as you controlled the Apple's built-in speaker.

Table 17–1 summarizes the four soft switches. As you can see, low-resolution graphics is selected when you access locations $C050 and $C056. The other two switches ($C052/3 and $C054/5) select either full-screen graphics or mixed graphics and text, and either page 1 or page 2, respectively. In this chapter, you will use full-screen graphics and page 1.

It doesn't matter what order you set these switches in, but the graphics/text switch should be set last, so that the other changes take place off-screen. Here are the switch-setting instructions that should be used in your programs:

TABLE 17–1. Screen Soft Switches

Location	Function
$C050	Select Graphics Mode.
$C051	Select Text Mode.
$C052	Select Full-Screen Display, Graphics or Text.
$C053	Select Graphics Plus Text. Text Is Displayed in a Four-Line "Window" at the Bottom of the Screen.*
$C054	Select Screen Page 1, Graphics or Text.
$C055	Select Screen Page 2, Graphics or Text.
$C056	Select Low-Resolution Graphics.*
$C057	Select High-Resolution Graphics.*

*If the screen is in the text mode, this will not be visible until location $C050 is accessed.

```
LDA   $C052   ;Select full-screen display
LDA   $C054   ; of page 1
LDA   $C056   ; in low-resolution graphics
LDA   $C050   ;Select graphics mode
```

Low-Resolution Graphics Subroutines

The Apple II monitor has several built-in subroutines you can use to program low-resolution graphics. They are summarized in Table 17–2.

The first three subroutines, PLOT, HLINE, and VLINE, are the primary "drawing" subroutines. Screen coordinates are passed to these subroutines in registers and memory locations. CLRSCR and CLRTOP clear the screen for full-graphics and graphics-plus-text displays, respectively. NEXTCOL, SETCOL, and SCRN are color control subroutines. Sixteen different colors, listed in Table 17–3, are available with low-resolution graphics. If you are using a black-and-white or a "green" display, use *white* as the display color. When programming a graphics-plus-text display, the SETGR subroutine is a convenient substitute for the four soft-switch instructions.

An Example Program

Figure 17–2 shows a program that plots a low-resolution diagonal line on the screen, between row 0, column 0 and row 39, column 39. Run this program on your Apple II to see what a typical low-resolution display looks like. The plotted line will remain on the screen until you press RESET.

TABLE 17–2. Low-Resolution Graphics Subroutines

Entry Point	Name	Function	Registers to Load Before Calling	Registers Affected
$F800	PLOT	Plot a Graphics Point on Low-Resolution Page 1.	A = Row Y = Column	A
$F819	HLINE	Draw a Low-Resolution Horizontal Line.	A = Row Y = Left Column Loc. $2C = Right Column	A and Y
$F828	VLINE	Draw a Low-Resolution Vertical Line.	Y = Column A = High Row Loc. $2D = Low Row	A
$F832	CLRSCR	Clear All 48 Low-Resolution Graphics Rows to Black. If in Text Mode, Fills Screen with @.	None	A and Y
$F836	CLRTOP	Clear the Upper 40 Rows, Leaving the Text Window Intact.	None	A and Y
$F85F	NEXTCOL	Increment the Current Low-Resolution Graphics Color by Three.	None	A
$F864	SETCOL	Set Low-Resolution Graphics Color.	A = Color Number	A
$F871	SCRN	Read the Color of a Low-Resolution Graphics Point.	A = Row Y = Column	A Contains Color Number
$FB40	SETGR	Set Low-Resolution Graphics Mode, Clear Screen, and Set Four-Line Text Window.	None	A and Y

Animating Low-Resolution Shapes

The program in Figure 17–2 produced a static display, one that does not move. In many applications, however, you'd like to move the shapes around the screen, that is, you'd like to *animate* them.

Producing animated effects is not as difficult as you might expect. In fact, it involves only five steps:

TABLE 17–3. Low-Resolution Graphics Colors

Number			Number		
Dec.	Hex.	Color	Dec.	Hex.	Color
0	$0	Black	8	$8	Brown
1	$1	Magenta	9	$9	Orange
2	$2	Dark Blue	10	$A	Grey #2
3	$3	Purple	11	$B	Pink
4	$4	Dark Green	12	$C	Light Green
5	$5	Grey #1	13	$D	Yellow
6	$6	Medium Blue	14	$E	Aquamarine
7	$7	Light Blue	15	$F	White

1. Plot the shape (or shapes) on the screen.
2. Wait some period of time, so that the shape is visible.
3. Erase the shape.
4. Change the shape's row and column parameters.
5. Repeat the process, starting at step 1.

There are two ways to erase a shape from the screen: You can either *clear* the entire screen or *replot* the shape in black. However, clearing the screen with the Monitor's CLRSCR subroutine makes the display blink on and off, so the second approach, replotting the shape in black, is more desirable. The easiest way to replot a shape is to put the plotting instructions in a subroutine. This allows you to call the same

```
        ORG   $300
        LDA   $C052    ;Select full-screen display
        LDA   $C054    ; of page 1
        LDA   $C056    ; in low-resolution graphics
        JSR   $F832    ;Clear screen with CLRSCR
        LDA   #15      ;Select the color white
        JSR   $F864    ; via SETCOL
        LDA   $C050    ;Select Graphics mode
START   LDY   #0       ;Start at column 0
PLOT    TYA            ; and row 0
        JSR   $F800    ;PLOT one point
        INY            ;Increment column
        CPY   #39      ;End of screen?
        BNE   PLOT     ;If not, go plot next point
        BEQ   START    ;If so, start over
```

FIGURE 17–2. Program to Plot a Diagonal Line

instructions twice, once to draw the shape (active color is white) and again to erase it (active color is black).

Figure 17–3 is the flowchart for the simplest kind of animation program, one that just moves a single graphic point across the screen. In this case, the point moves across row 20, from column 0 to column 39, and is displayed for 0.3 second at each column position.

Figure 17–4 shows the source code for the point-moving program. The first four lines are *equate directives* that assign names to four Monitor subroutines used in the program. This allows you to reference them by name rather than entry point, thereby making your typing job a little less error prone as well as making the source code easier to read and understand.

The equates are followed by the usual graphics initialization instructions and then the main program, START. Note that the 0.3-second display delay is produced by calling the Monitor's WAIT subroutine three times.

The point-drawing subroutine is called DRAW. Notice it ends with a jump instruction (to the Monitor's PLOT subroutine) rather than with an RTS instruction. This is possible because PLOT, being a subroutine itself, is terminated with an RTS. When PLOT's RTS is executed, the 6502 will return directly to the main program instead of to DRAW. In fact, *any* subroutine ending with a subroutine call can be terminated with a JMP instruction instead of with a JSR-then-RTS combination.

Following DRAW are the two define byte (DFB) directives that store the initial row and column parameters into memory and assign names to these locations. Every low-resolution graphics program you write should have a comparable table, one that has a row and column coordinate for each point in the shape you wish to display.

After executing the program as listed here, you should experiment with a few different display periods, to make the point move faster or slower across the screen. The instruction that determines the display period is

```
LDX  #3
```

The 3 operand produces a 0.3-second display. A higher number will make the point move slower; a lower number will make the display move faster. Therefore, to change the speed, you must change the second byte of this LDX instruction. If you followed Figure 17–4 exactly, that byte will be stored at location $318. Thus, your change command should be of the form

```
*318:hh
```

You must also reinitialize the COLUMN coordinate to zero. It should be at location $341. With these two changes made, you can execute the altered program with *300G.

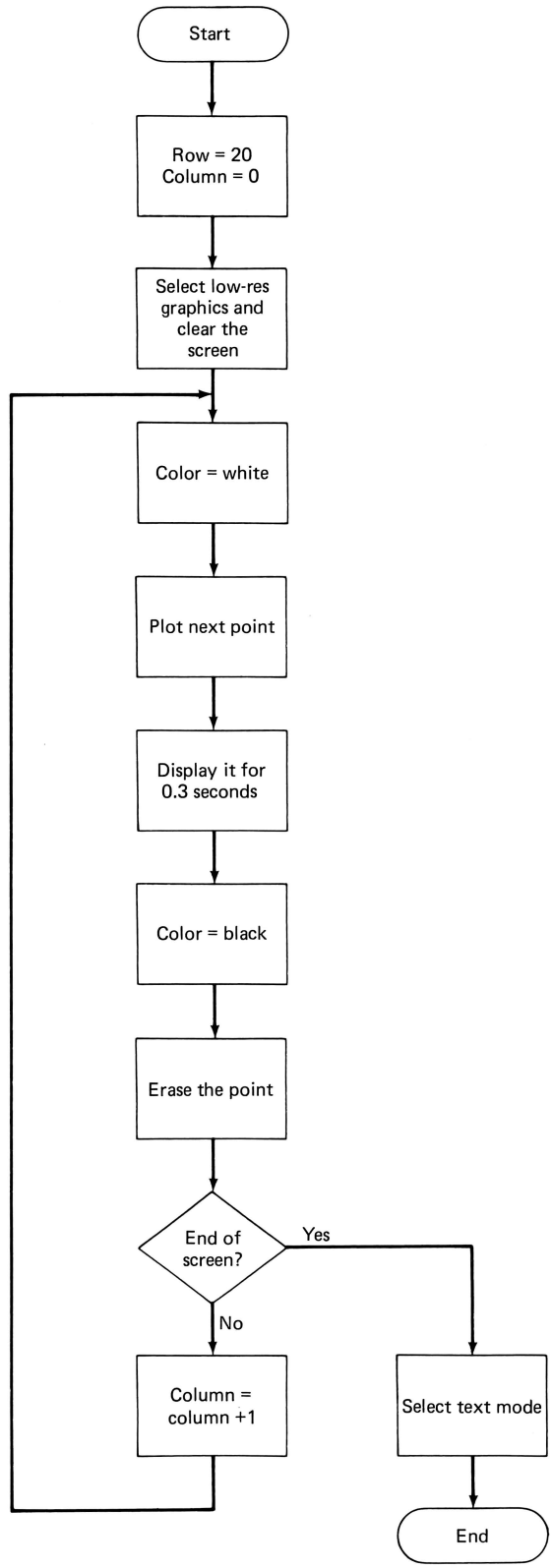


FIGURE 17–3. Flowchart for Program to Move a Graphics Point Across the Screen

```
*Equates to reference Monitor subroutines by name
*
PLOT    EQU  $F800       ;Plot a graphics point
CLRSCR  EQU  $F832       ;Clear the screen
SETCOL  EQU  $F864       ;Set the color
WAIT    EQU  $FCA8       ;Wait a specified interval
*
*Initialize low-resolution graphics
*
        ORG  $300
        LDA  $C052       ;Select full-screen display
        LDA  $C054       ; of page 1
        LDA  $C056       ; in low-resolution graphics
        JSR  CLRSCR      ;Clear the screen
        LDA  $C050       ;Select Graphics mode
*
*Main program starts here
*
START   LDA  #15         ;Set color to white
        JSR  SETCOL
        JSR  DRAW        ; and draw next point
        LDX  #3          ;Display it for 0.3 seconds
DELAY   LDA  #200
        JSR  WAIT
        DEX
        BNE  DELAY
        LDA  #0          ;Set color to black
        JSR  SETCOL
        JSR  DRAW        ; and erase the point
        CPY  #39         ;End of screen?
        BEQ  BREAK       ;If so, exit
        INC  COLUMN      ;If not, move to next column
        JMP  START       ; and start over
BREAK   LDA  $C051       ;Select Text mode
        BRK              ; and return to Monitor
*
*This subroutine draws the graphics point.
*
DRAW    LDA  ROW         ;Fetch row number
        LDY  COLUMN      ; and column number,
        JMP  PLOT        ; then PLOT the point
*
*Initialize row and column locations
*
ROW     DFB  20          ;Row = 20
COLUMN  DFB  0           ;Column = 0
```

FIGURE 17–4 The Point-Moving Program

EXERCISES

1. You are to apply your newly acquired knowledge of low-resolution graphics to develop a program that displays a 7-column by 6-row rectangle on the screen. The rectangle should be displayed continuously, until you press RESET.
   The top of the rectangle should lie in row 20, the bottom in row 25. The left side of the rectangle should lie in column 0, the right side in column 6.
   a. Draw a flowchart of the rectangle-plotting program below.

b. Write the source code for the program in the space below. Use the Monitor subroutines HLINE and VLINE to plot the horizontal and vertical sides of the rectangle, respectively. After writing the source code, enter and run it on your Apple II system.

2. Now develop a program that moves the rectangle left to right across the screen. With each move, the rectangle should be displayed for 0.3 second.

   Since your flowchart will resemble the one in Figure 17–3 (except you are plotting a rectangle instead of a point), a new flowchart is unnecessary. Write the source code for the rectangle-moving program in the space below, then run the program.

CHAPTER EIGHTEEN

Reset and Interrupts

OBJECT

To understand the reset and interrupt characteristics of the 6502 microprocessor and their application in the Apple II microcomputer.

NEW INSTRUCTIONS

This chapter describes the interrupt-related instructions CLI, SEI, and RTI.

PREPARATION

Read all material referenced under the headings "RESET" and "interrupts" in the Index to the *Apple II Reference Manual.*

DISCUSSION

Being essentially personal/business computers, most leading 6502-based microcomputers have similar features. All have a built-in keyboard (or numeric keypad); all have, or can be connected to, a display, a printer, and a cassette recorder or floppy disk drive; most come with some form of BASIC; and so on. However, the operating programs, or *monitors,* that drive these various computers are quite dissimilar. You could not plug a PET Monitor chip into an Apple II, for instance, and expect the Apple to operate properly, because the Apple's circuitry is different from the PET's circuitry.

Despite their dissimilarities, all these computers use the 6502 as their microprocessor and, consequently, have at least the following in common:

In every 6502-based system, the highest 6 bytes in memory ($FFFA through $FFFF) serve a dedicated function: they hold the starting addresses of the system's reset and interrupt service routines.

As you know from the preparatory reading, the 6502 has one reset line ($\overline{RES}$) and two interrupt lines, interrupt request ($\overline{IRQ}$) and nonmaskable interrupt ($\overline{NMI}$). The starting addresses of the service routines for these three lines differ between the Standard Monitor and the Autostart Monitor. Table 18–1 summarizes these differences.

TABLE 18–1. Apple II Reset and Interrupt Vectors

		Address Held in Vector Locations	
Vector	Vector Locs.	Standard Monitor	Autostart Monitor
$\overline{NMI}$	$FFFA, $FFFB	$03FB	$03FB
$\overline{RES}$	$FFFC, $FFFD	$FF59	$FA62
$\overline{IRQ}$	$FFFE, $FFFF	$FA86	$FA40

Reset

The 6502 microprocessor has a reset line, labeled $\overline{RES}$, that is activated when you turn the power on or press RESET. In response to $\overline{RES}$, the 6502 automatically initiates a six-cycle *start sequence* during which two things happen:

1. The IRQ disable bit (I) in the status register is set to 1, to "lock out" external interrupts while the microprocessor is being initialized.
2. The program counter is loaded with the address contained in locations $FFFC (low byte) and $FFFD (high byte).

Because this occurs when power is first turned on, locations $FFFC and $FFFD must be in read-only memory (ROM), rather than in read/write memory (which contains "garbage" at power-up time). Naturally, the program addressed by the contents of $FFFC/$FFFD must also be in ROM, so that something meaningful occurs when you turn on the power.

In the Apple II, the program addressed by $FFFC/$FFFD (the *system reset routine*) puts the screen in the text mode, loads each of the 6502's registers to some initial value, and performs other miscellaneous "housekeeping" tasks. One such task is to call the proper language. The Standard Monitor always calls the Monitor; the Autostart Monitor calls Applesoft BASIC.

In some of the preceding exercises, you manually initiated a reset by pressing the RESET key. The *Apple II Reference Manual* distinguishes between these two different kinds of resets by referring to the power-up reset as a "cold reset" and to the RESET-key reset as a "warm reset." A warm reset allows you to regain control of the computer if a program has gone wild or has entered an endless loop (planned or accidentally).

A warm reset, using the RESET key, leaves all programs and variables intact, but causes a return to the "start" language, either BASIC or Monitor. Page 36 of the *Apple II Reference Manual* describes how to change three bytes in memory to cause an Autostart system to be reset to the Monitor rather than to BASIC.

Interrupt Request ($\overline{\text{IRQ}}$)

The 6502 microprocessor has a line called $\overline{\text{IRQ}}$ (interrupt request) that permits external devices to request servicing from the microprocessor. The $\overline{\text{IRQ}}$ line acts very similar to the bell on a telephone. Just as a telephone rings to inform you that someone is calling (that is, someone is requesting our attention), the $\overline{\text{IRQ}}$ line informs the microprocessor that some device in the system—perhaps a floppy disk drive—needs something. In most cases, the "something" is data. And just as you can choose to ignore a ringing telephone, the microprocessor can ignore an $\overline{\text{IRQ}}$ interrupt. The 6502 will ignore interrupts whenever the interrupt disable (I) bit in the status register is set to 1 and will respond to interrupts when I is cleared to 0.

The 6502 has two instructions that can be used to set or clear the I bit:

Instruction	Description
CLI	Clear Interrupt Disable Bit (Accept $\overline{\text{IRQ}}$s)
SEI	Set Interrupt Disable Bit (Ignore $\overline{\text{IRQ}}$s)

If the interrupt disable bit (I) is clear and some external device activates the $\overline{\text{IRQ}}$ line (pulls it low), the 6502 finishes executing the current instruction and then automatically initiates an eight-cycle *interrupt sequence*. During this sequence, the 6502 does the following:

1. Pushes three bytes of "return information" onto the stack. These bytes are the high and low bytes of the program counter and the contents of the processor status register.
2. Loads the program counter with the address contained in locations $FFFE (low byte) and $FFFF (high byte).
3. Sets the interrupt disable bit (I) in the status register to temporarily lock out subsequent interrupt requests.

In the Apple II Standard Monitor, locations $FFFE and $FFFF "point to" location $FA86; in the Autostart Monitor, these locations point to location $FA40. In either case, the "pointed-to" location is the starting location of the Apple's interrupt-handling routine. This routine, called IRQ, is similar to a subroutine but ends with a return from interrupt (RTI) instruction rather than with a return from subroutine (RTS) instruction.

Figure 18–1 shows how the stack pointer (S), the stack, the program counter (PC),

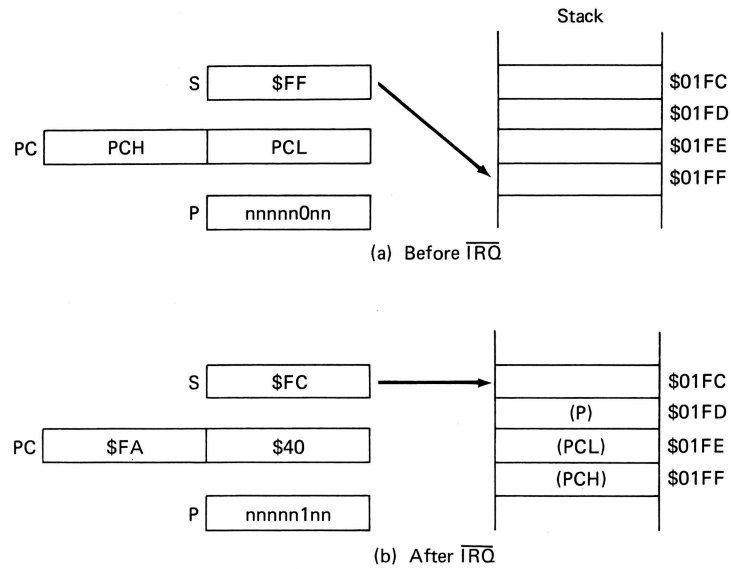


FIGURE 18–1. Effects of $\overline{\text{IRQ}}$ on an Apple II (Autostart Monitor)

and the processor status register (P) are affected by the $\overline{\text{IRQ}}$ interrupt for an Apple with the Autostart Monitor. For an Apple with the Standard Monitor, the final address in the PC would be $FA86 rather than $FA40.

The 6502 microprocessor is designed so that a break (BRK) instruction causes program control to the same handling routine as an $\overline{\text{IRQ}}$. Therefore, the interrupt-handling routine must determine whether the handling routine was called due to an interrupt or a BRK instruction. It makes this determination based on the state of the break command (B) flag in the processor status register. If B is set, the call was due to a BRK instruction; if B is clear, the call was due to an $\overline{\text{IRQ}}$.

None of the circuitry in the Apple II is interrupt-driven, so that the Monitor does not contain instructions to process a specific $\overline{\text{IRQ}}$ interrupt. Instead, if the processor determines that an *interrupt* caused the interrupt-handling routine to be called, the 6502 simply executes a

```
JMP   ($03FE)
```

instruction. That is, program control is transferred to the address contained in memory locations $03FE (low byte) and $03FF (high byte).

Because the Apple II itself does not use $\overline{\text{IRQ}}$ interrupts, these locations, $03FE and $03FF, are initialized by DOS to point to the Monitor routine, MON, which starts at location $FF65. If you add any interrupt-driven hardware to the system, $03FE/$03FF must be changed to the address of *your* interrupt-handling routine.

Nonmaskable Interrupt ($\overline{\text{NMI}}$)

The other interrupt line of the 6502 microprocessor, nonmaskable interrupt ($\overline{\text{NMI}}$), provides interrupts for high-priority events that cannot be ignored, such as power failure. Unlike $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$-generated interrupts cannot be masked out or disabled. The 6502 always processes an $\overline{\text{NMI}}$ interrupt immediately after the current instruction has been executed.

In a 6500 system, an $\overline{\text{NMI}}$ interrupt always has higher priority than an $\overline{\text{IRQ}}$ interrupt. If the two occur simultaneously, the microprocessor processes the nonmaskable interrupt first. Aside from these differences, the operation of $\overline{\text{NMI}}$ differs very little from that of $\overline{\text{IRQ}}$. For $\overline{\text{NMI}}$, the vector pointer is fetched from locations $FFFA (low-address byte) and $FFFB (high-address byte), rather than from $\overline{\text{IRQ}}$ locations $FFFE and $FFFF. In fact, Figure 18–1 also illustrates how the 6502 microprocessor responds to $\overline{\text{NMI}}$, except that for $\overline{\text{NMI}}$ the postinterrupt value of the program counter would be $03FB rather than $FA40.

None of the circuits in the Apple II use $\overline{\text{NMI}}$, so locations $03FB, $03FC, and $03FD hold a JMP instruction to the monitor routine, MON. In both the Standard Monitor and the Autostart Monitor, this command is

```
JMP   $FF65
```

Therefore, in an unaltered Apple II (any model), an $\overline{\text{IRQ}}$ or any $\overline{\text{NMI}}$ will cause the Monitor to be called.

EXERCISES

1. Determine the effects of the RESET key on the 6502's internal register by initializing the registers to the following values:

   A = 00 X = 00 Y = 00 P = 20 S = FF

   then pressing the RESET key.

   Now, examine the register contents again, and record your findings below.

   A = X = Y = P = S =

2. To what values are the registers initialized when you turn on the power? Find out by turning off the power, then turning it on again. Examine the registers after power-up and record your findings below.

   A = X = Y = P = S =

3. If you have an Apple II Plus, or an Apple II with the Applesoft II Firmware card installed, pressing RESET will put the computer in BASIC. Use the procedure described on page 37 of the *Apple II Reference Manual* to cause RESET to call up the Monitor.

4. Without connecting external circuitry to the Apple II, there is no way to actually exercise the $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$ interrupts. However, it is certainly possible to *simulate* these interrupts with software in order to give a better understanding of how they operate.

   To stimulate an $\overline{\text{IRQ}}$ interrupt, develop a program that counts the number of times the I key is pressed. Each time that key is pressed, the program should perform an indirect jump through the dedicated $\overline{\text{IRQ}}$ vector ($FFFE/$FFFF), thereby transferring control to the Apple's $\overline{\text{IRQ}}$-handling routine. Since this "interrupt" is not due to a BRK instruction, the handling routine will transfer control to the address contained in the user routine vector, locations $03FE and $03FF.

   In your program, the user routine vector should contain the address of a routine that increments a count register (use X). The last instruction in the counting routine should be a JMP back to the calling program. Counting should continue until the S (for stop) key is pressed, at which point a BRK instruction terminates the program. Naturally, all keys except I and S should be ignored.

   a. Draw the flowchart of the main program and the counting routine below.

b. Write the code for the main program and the counting routine below.

5. Enter the main program into memory starting at location $300, then enter the counting routine into memory starting at location $1000. This done, run the program for various numbers of key depressions to make sure it works.

Some Closing Comments

With the final exercise completed, you have every right to feel a sense of pride in your accomplishments. Starting with the most fundamental instructions, load and store, you progressed through the 6502's entire instruction set. Along the way, you developed a series of "utility" programs, including those that can be used to multiply and divide numbers, to process data in tables and lists, and to convert ASCII input to binary (and vice versa).

With this foundation, you explored the Apple II's sound-generating capabilities, from single tones to a two-octave music program. You then developed low-resolution graphics programs and mastered the rudiments of animation. Finally, you investigated the Apple II's reset and interrupt characteristics as a background for programming expanded systems.

Many instruction sequences and subroutines you developed in this book can be used unaltered, or with minor changes, in future programs. More important, however, you now have the background to quickly and confidently develop programs of your own design. And because you are using assembly language, and using it *efficiently*, you have the satisfaction of knowing that the full power and flexibility of the Apple II are now at your command.

Answers to Exercises

Following are solutions to the problems in the Exercise section of each chapter. All answers given are those proposed by the author. Some of your answers—particularly programs and flowcharts—may differ from those printed here without necessarily being incorrect. Because this is an educational book, the best solution is one that not only solves the problem, but also helps you learn the *principles* being studied.

CHAPTER 1. GETTING INSIDE THE APPLE II

1. When you turn on the power,

   A = $98

   (This is the value obtained from the Apple II Plus used to prepare this book; your value may differ. The main point is that the contents of A, and every other register, are *random*. Don't assume they contain some predictable value.)
2. The CTRL-E command is used to examine registers.
3. The current contents of location $10 are 00. Again, this value may be different in your computer. Like register contents, memory contents are initially random.
4. To determine the contents of location $10, use the command

   *10 (RETURN)

5. a. The completed table should look like this:

Instruction	Hex. Values
LDA #$AC	$A9
	$AC
STA $10	$85
	$10
BRK	$00

   b. The load instruction uses immediate addressing.
   c. The store instruction uses zero page addressing.
   d. The break instruction uses implied addressing.

6. The sequence to enter the program into memory is

```
*300:A9 AC 85 10 00
```

7. The disassembly command is

```
*300L
```

   It displays twenty instructions, but you are only interested in the first three.

8. To execute the program , use this command:

```
*300G
```

9. The final contents of the Program Counter, Accumulator, and location $10 are:

```
PC = $0306  A = $AC  ($10) = $AC
```

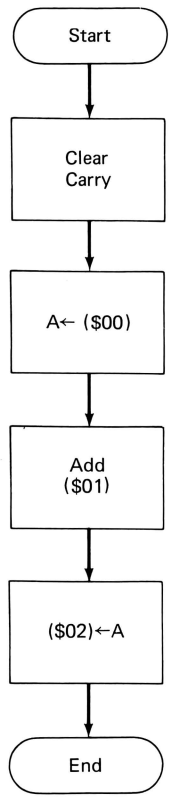
   Remember, the BRK instruction always produces a Program Counter value two bytes past the BRK instruction. Since BRK occupies memory location $304, PC ends up with the value $306. If you entered

```
*G  (RETURN)
```

   right now, the 6502 would start executing at location $306, with unpredictable results.

CHAPTER 2. ADDITION OPERATIONS

1. a. The flowchart looks like this:



b. The program to perform the add operation is

```
CLC       ;Clear Carry
LDA 0     ;Load $00
ADC 1     ;Add contents of $01 to it
STA 2     ; and store the result
BRK
```

c. The results for the test cases are:

$00		$01		$02		
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	Carry? (Yes/No)
143	8F	131	83	274	12	Yes
46	2E	65	41	111	6F	No
59	3B	197	C5	256	00	Yes

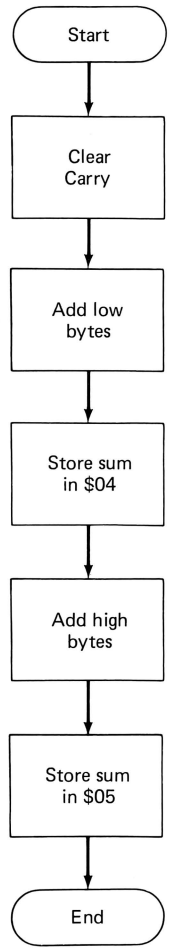
The answers are invalid in the first and third cases because a single location cannot hold a value that exceeds 255.

2. a. The program to add two signed numbers will be no different from the program to add unsigned numbers. To the 6502 microprocessor, both types of numbers are simply values in memory. It's up to you to interpret what kinds of numbers are being added.

$00		$01		$02		
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	Overflow? (Yes/No)
106	6A	89	59	195	C3	Yes
127	7F	3	03	130	82	Yes
32	20	–3	FD	29	1D	No
–90	A6	–62	C2	–152	68	Yes

b. All three answers that have overflow are invalid because a single memory location cannot hold a signed value that exceeds +127 or –128.

3. a. The flowchart for the double-precision signed add looks like this:



b. The program is

```
CLC      Clear Carry
LDA 0    ;Add low bytes
ADC 2
STA 4    ; and store the result
LDA 1    ;Add high bytes
ADC 3
STA 5    ; and store the result
BRK
```

c. The answers for the test run are:

Addend #1		Addend #2		Sum		
$00	$01	$02	$03	$04	$05	Overflow? (Yes/No)
$63	$25	$AB	$41	$0E	$67	No
$C7	$83	$B4	$F1	$7B	$74	Yes

CHAPTER 3. SUBTRACTION AND LOGICAL OPERATIONS

1. The 24-bit subtraction program is:

```
SEC       ;Set Carry for subtraction
LDA 3     ;Subtract low-order bytes
SBC 0
STA 3     ; and store the result
LDA 4     ;Subtract middle bytes
SBC 1
STA 4     ; and store the result
LDA 5     ;Subtract high-order bytes
SBC 2
STA 5     ; and store the result
BRK
```

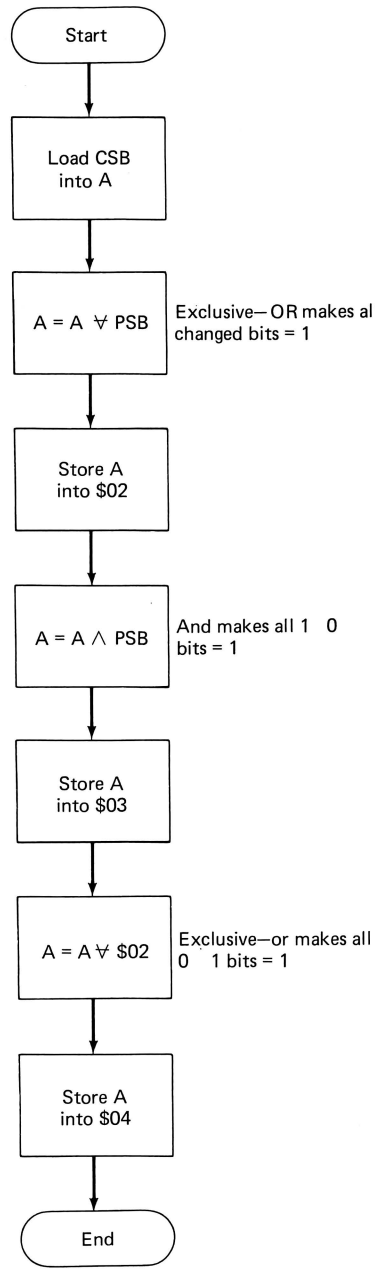
2. a. Results of subtracting $74A017 from $267BFE are

Subtrahend			Minuend			Results			
$00	$01	$02	$03	$04	$05	$03	$04	$05	Carry =
17	A0	74	FE	7B	26	E7	DB	B1	0

b. Carry = 0 in the result indicates that the result is negative; that is, the subtract operation required a borrow.

c. By checking the final state of overflow, find out whether the result is valid. Here, V = 0, so the result is valid.

3. a. The flowchart for the industrial control program is:


Start
Load CSB into A
A = A ∀ PSB
Exclusive–OR makes all changed bits = 1
Store A into $02
A = A ∧ PSB
And makes all 1 0 bits = 1
Store A into $03
A = A ∀ $02
Exclusive–or makes all 0 1 bits = 1
Store A into $04
End

b. The source code for the industrial control program is:

```
LDA 0     ;Load CSB into A
EOR 1     ;Mask out all but changed bits
STA 2     ; and store this value into $02
AND 1     ;Set bits that changed from 1 to 0
STA 3     ; and store this value into $03
EOR 2     ;Set bits that changed from 0 to 1
STA 4     ; and store this value into $04
BRK
```

4. a. The correct entries in Table 3–5 are:

Device	PSB	CSB
Pressure Sensor	Above Setpoint	Above Setpoint
Temperature Sensor	Above Setpoint	Above Setpoint
Velocity Sensor	Above Setpoint	Above Setpoint
Flow Rate Sensor	At or Below Setpoint	At or Below Setpoint
Concentration Sensor	Above Setpoint	Above Setpoint
Valve A	Closed	Closed
Valve B	Closed	Open
Power	On	Off

b. The following changes occurred:
Devices that changed: valve B and power
Devices changed from 1 to 0: power
Devices changed from 0 to 1: valve B

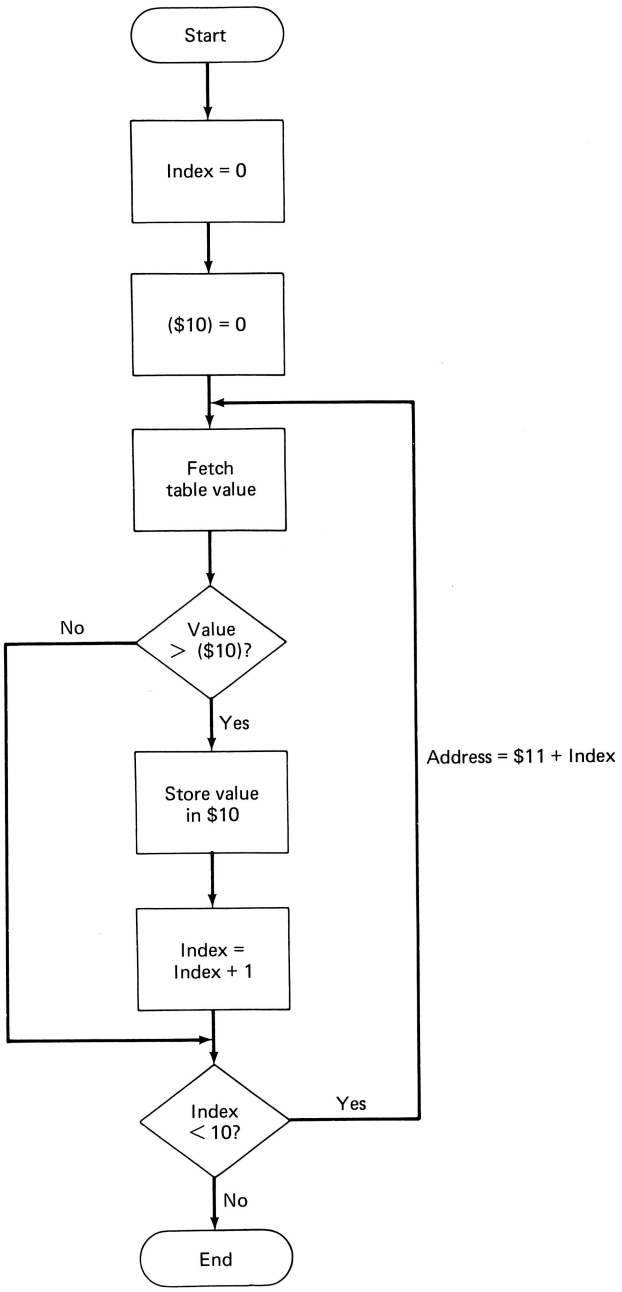
CHAPTER 4. PROGRAM SEQUENCING

1. Here is a program that follows the flowchart in Figure 4–1:

```
            LDA   $11       ;Load first value
            CMP   $10       ;Values out of order?
            BCC   SWAP
            BEQ   EQUAL     ;No.   $10 is
            LDA   #0        ; less than $11
            JMP   STORE
EQUAL       LDA   #1        ; or equal to $11
            JMP   STORE
SWAP        LDX   $10       ;Yes.  $10 is greater than $11
            STX   $11       ;Exchange $10 and $11
            STA   $10
            LDA   #2
STORE       STA   $12       ;Store indicator in $12
            BRK
```

2. For the test runs, the contents of $12 are 2, 1, 0, and 1, respectively.

3. a. The flowchart for the maximum value program is:



b. The code for the maximum value program is:

```
        LDX  #0       ;Index = 0
        STX  $10      ;Result location = 0
LOOP    LDA  $11,X    ;Fetch next table value
        CMP  $10      ;Is it greater than ($10)?
        BCC  UPIND
        BEQ  UPIND
        STA  $10      ;Yes.  Change location $10
UPIND   INX           ;Increment index
        CPX  #10      ;Finished with table?
        BCC  LOOP     ;No.  Go back for next value
        BRK           ;Yes.  Exit
```

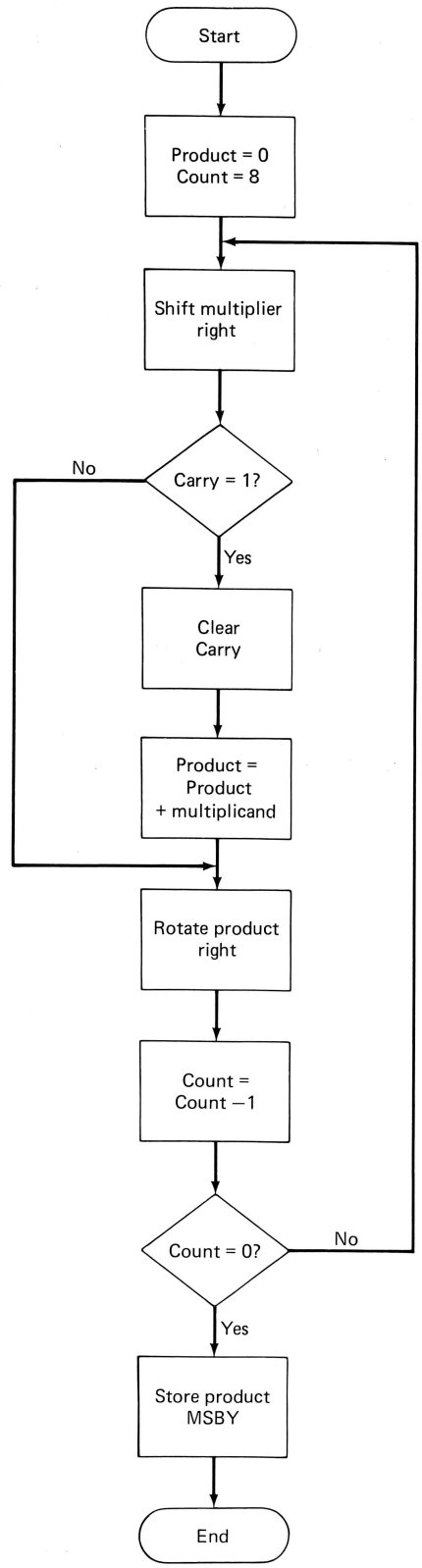
4. If the program works correctly,

   Largest number ($10) = $FA

CHAPTER 6. MULTIPLICATION OPERATIONS WITH SHIFT AND ROTATE

1. The results should be the same as shown in the discussion. That is, ASL should yield $68, LSR should yield $1A, ROL should yield $69, and ROR should yield $9A.

2. a. The flowchart for the single-precision multiplication is



b. The multiplication program should look like this:

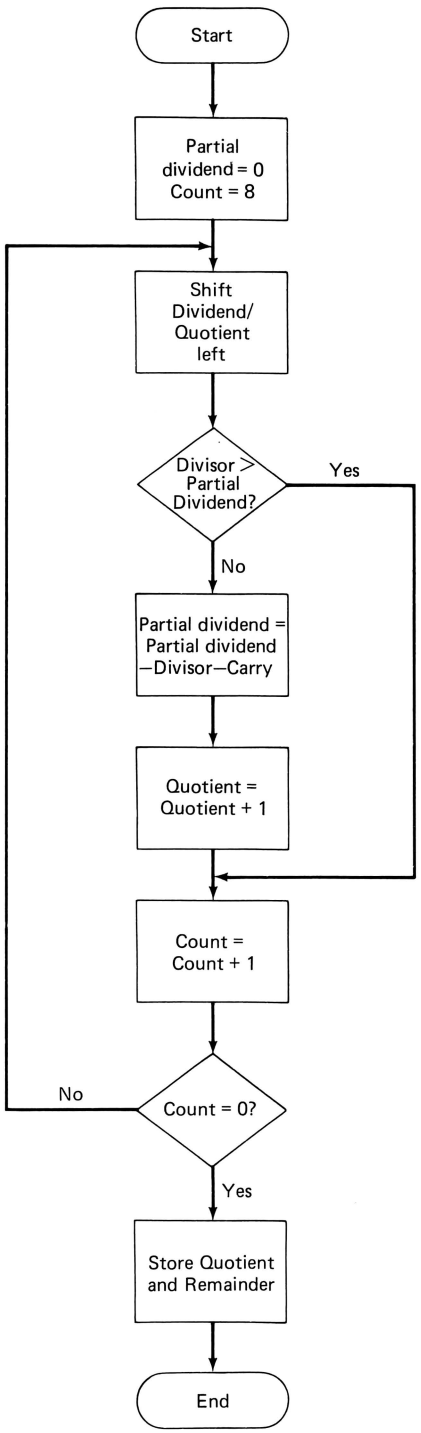
```
        LDA   #0        ;Clear MSBY of product
        LDX   #8        ;Multiplier bit count = 8
NXTBT   LSR   $10       ;Get next multiplier bit
        BCC   ALIGN     ;Multiplier bit = 1?
        CLC             ;Yes.  Add multiplicand
        ADC   $11
ALIGN   ROR   A         ;Rotate partial product right
        ROR   $12
        DEX             ;Decrement bit count
        BNE   NXTBT     ;Loop until 8 bits are done
        STA   $13       ; then store product MSBY
        BRK
```

3. The results for the four test cases are:

Multiplier $10		Multiplicand $11		Product $12, $13	
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.
255	FF	1	01	255	00FF
103	67	124	7C	12772	31E4
255	FF	0	0	0	0
255	FF	255	FF	65025	FE01

CHAPTER 7. DIVISION OPERATIONS

1. a. The flowchart for the single-precision division is:



b. The division program should look like this:

```
        LDA  #0      ;Clear partial dividend
        LDX  #8      ;Dividend bit count = 8
NXTBT   ASL  $11     ;Shift dividend/quotient left
        ROL  A       ; into partial dividend
        CMP  $10     ;Compare divisor to partial dividend
        BCC  CNTDN   ;Divisor greater than partial di-
                     vidend?
        SBC  $10     ;No.  Subtract divisor
        INC  $11     ; and set bit in quotient
CNTDN   DEX          ;Decrement bit count
        BNE  NXTBT   ;Loop until 8 bits are done
        STA  $12     ; then store remainder
        BRK
```

Note that it was unnecessary to set the carry prior to the subtract, since it *must* have been set for BCC CNTDN to "fall through" to the subtract instruction.

2. The results of the four test cases are:

Divisor $10		Dividend $11		Quotient $11		Remainder $12	
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	Dec.	Hex.
1	01	255	FF	255	FF	0	00
15	0F	170	AA	11	0B	5	05
255	FF	1	01	0	00	255	FF
254	FE	255	FF	1	01	1	01

3. Detecting a zero divisor is simply a matter of comparing the divisor to zero before the division operation begins. This comparison can be made by adding just two instructions to the program, immediately following LDA #00. They are

```
CMP  $10
BEQ  DONE
```

where DONE references the BRK instruction.

CHAPTER 8. SUBROUTINES AND THE STACK

1. The stack program is, simply:

```
LDX  #$FF      ;Put stack address into X
TXS            ; and transfer it to the Stack Pointer
LDA  #1        ;Initialize the registers
LDX  #2
LDY  #3
PHA            ;Push Accumulator
TXA            ;Push X Register
PHA
TYA            ;Push Y Register
PHA
BRK
```

Note that all push operations must take place via the accumulator. This also tells you that if the accumulator is to be saved (as it must here), the accumulator must be pushed onto the stack *first*.

2. a. The before and after "snapshots" of the stack are:

Before	Location	After
xx	$01FB	xx
xx	$01FC	xx
xx	$01FD	$03
xx	$01FE	$02
xx	$01FF	$01

The xx symbols in the table indicate that you don't care what the contents are. They are assumed to be random.

b. PHA stores accumulator in memory location $01FC.

c. PLA pulls accumulator from memory location $01FD.

3. a. The additional instructions serve to restore the original contents of A, X, and Y. They are:

```
LDA  #0        ;Load registers with zero
LDX  #0
LDY  #0
PLA            ;Pull Y Register
TAY
PLA            ;Pull X Register
TAX
PLA            ;Pull Accumulator
```

   Note that the registers are pulled in the reverse order from which they were pushed. Incidentally, the LDX #0 and LDY #0 could be replaced with TAX and TAY, respectively, causing the program to occupy less space in memory.

   b. The register contents should be:

   Accumulator = $01 X Register = $02 Y Register = $03

4. a. This subroutine adds 5 to location $10 and saves the contents of the accumulator:

```
PHA            ;Save A on the stack
CLC            ;Add 5 to location $10
LDA  $10
ADC  #5
STA  $10
PLA            ;Restore contents of A
RTS            ;Return
```

   b. The calling program is:

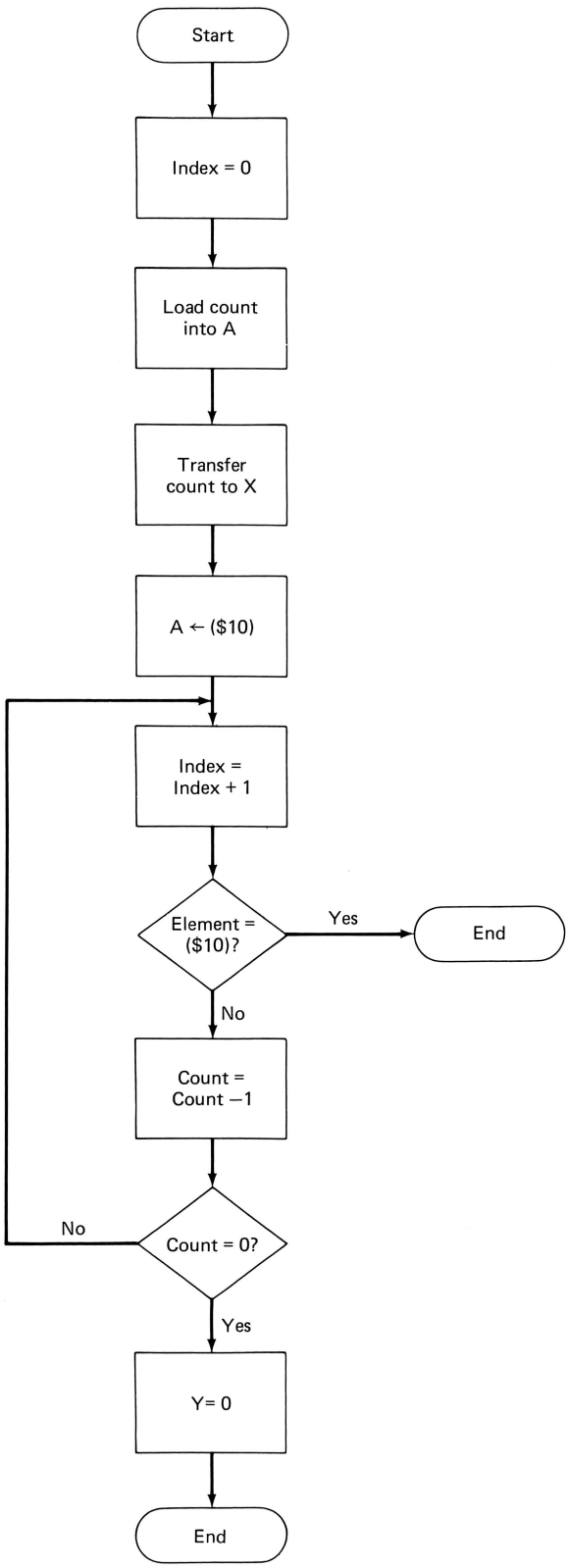
```
LDA  #0        ;Initialize location $10 to zero
STA  $10
JSR  $300      ;Call subroutine three times
JSR  $300
JSR  $300
BRK
```

5. After adding 5 to location $10 three times:

   Location $10 = $0F

CHAPTER 9. UNORDERED LISTS

1. The flowchart for the search program is shown below.



2. This program will return the element number in Y if the search value is found in the list and will return Y = 0 if the search value is not found in the list:

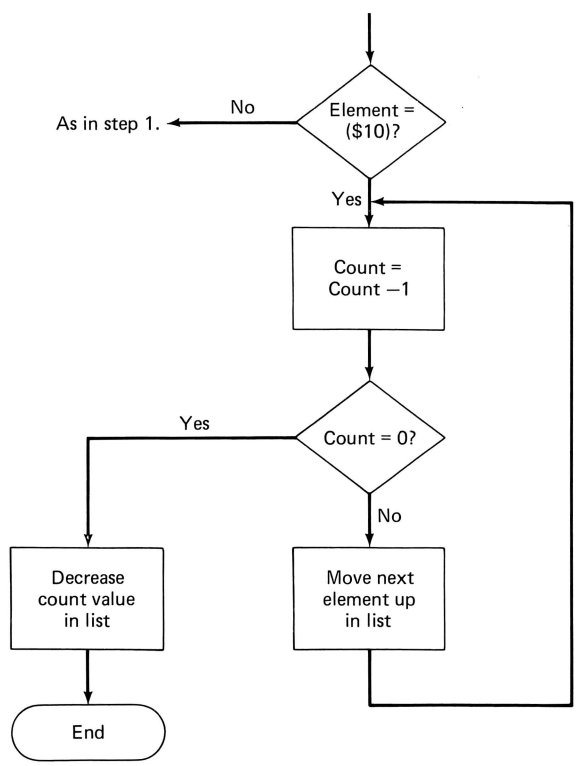
```
        LDY   #0        ;Fetch element count
        LDA   ($11),Y
        TAX             ; and put it in X
        LDA   $10       ;Load search value into A
NEXT    INY             ;Index to next element
        CMP   ($11),Y   ;Element = ($10)?
        BEQ   DONE      ;Yes.  Return with index in Y
        DEX             ;No.  Decrement element count
        BNE   NEXT      ;More elements to compare?
        LDY   #0        ;No.  Value was not found
DONE    BRK
```

3. a. The correct initial values are

Search value	($10) = $55
List address, low	($11) = $00
List address, high	($12) = $10

   b. Since $55 is the eighth data element in the list, the Y register should contain $08.

4. The portion of the flowchart that deletes an element from the list looks like this:



5. A program that will perform the search-and-destroy mission is

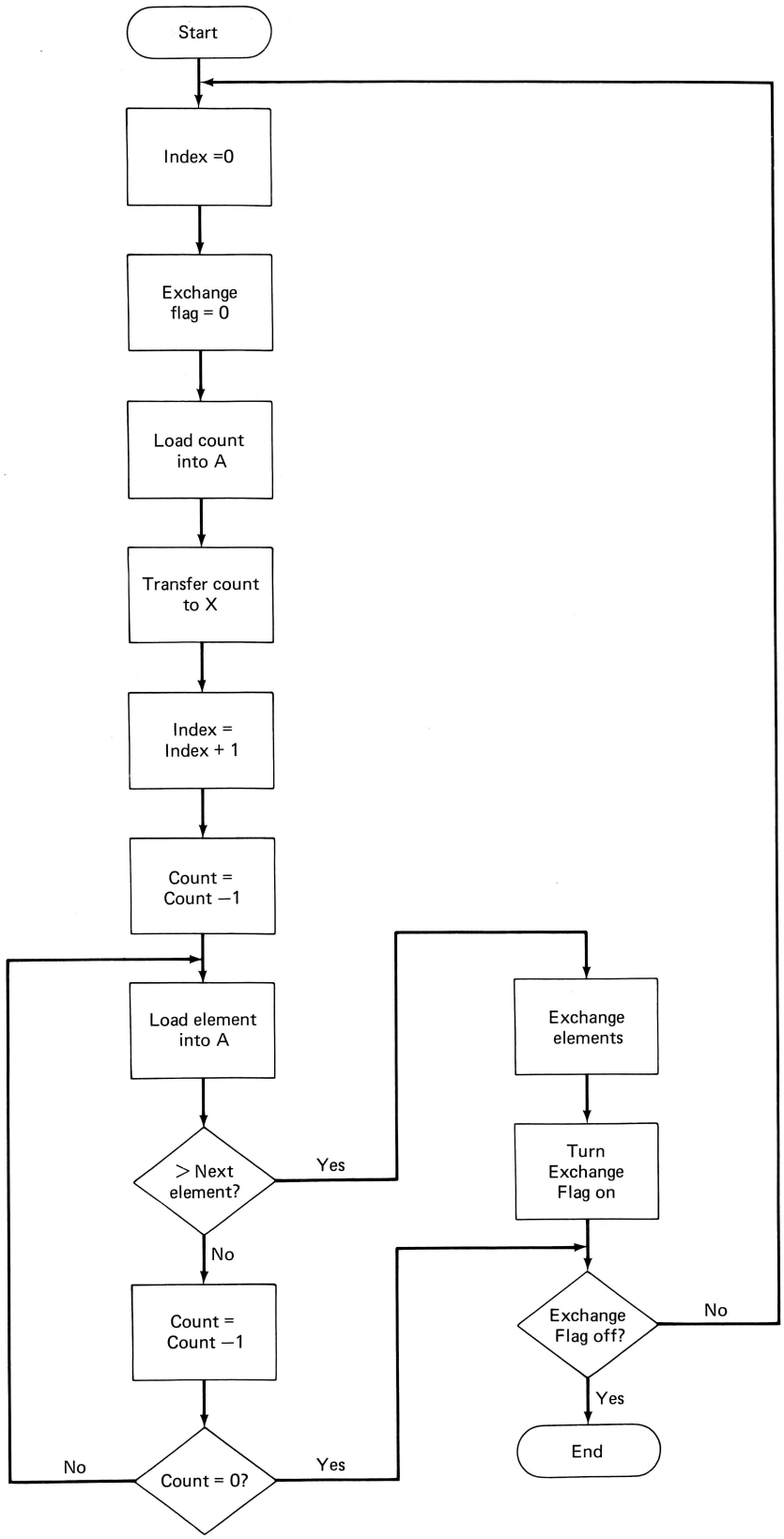
```
            LDY   #0          ;Fetch element count
            LDA   ($11),Y
            TAX               ; and put it into X
NEXT        INY               ;Index to next element
            CMP   ($11),Y     ;Element = ($10)?
            BEQ   KILL        :Yes.  Delete this element
            DEX               ;No.  Decrement element count
            BNE   NEXT        ;More elements to compare?
            BRK               ;No.  Return to Monitor
KILL        DEX               ;Decrement element count
            BEQ   DECCNT      ;End of list?
            INY               ;No.  Move next element up
            LDA   ($11),Y
            DEY
            STA   ($11),Y
            INY
            JMP   KILL
DECCNT      LDA   ($10,X)     ;Decrease element count in list
            SBC   #1
            STA   ($10,X)
            BRK
```

6. The search value is contained in location $1008 of the original list. After it has been deleted, the requested locations have these values:

($1000) = $09 (Element count is now 9, rather than 10)
($1007) = $FA (No change)
($1008) = $06
($1009) = $4F
($100A) = $4F (No longer part of the list)

CHAPTER 10. SORTING UNORDERED DATA

1. a. The bubble sort flowchart should resemble this one:



b. The bubble sort program is:

```
SORT8   LDY   #0        ;Set index = 0
        STY   $13       ; and turn off exchange flag
        LDA   ($11),Y   ;Fetch element count
        TAX             ; and put it into X
        INY             ;point to first element
        DEX             ;Decrement count
NEXT    LDA   ($11),Y   ;Fetch element
        INY
        CMP   ($11),Y   ;Is it larger than the next element?
        BCC   CHKEND
        BEQ   CHKEND
*                       Yes.  Exchange elements
        PHA             ;Save low byte on the stack
        LDA   ($11),Y   ; then get high byte
        DEY             ; and store it at low address
        STA   ($11),Y
        PLA             ;Pull low byte from stack
        INY             ; and store it at high address
        STA   ($11),Y
        DEC   $13       ;Turn on exchange flag (= -1)
CHKEND  DEX             ;End of list?
        BNE   NEXT      ;No.  Fetch next element
        BIT   $13       ;Yes.  Exchange flag still off?
        BMI   SORT8     ;No.  Go through list again
        BRK             ;Yes.  List is now sorted
```

2. When this list has been sorted, it should be in the following order:

Location	Contents	
$1000	$0A	(Element count should be unchanged)
$1001	$00	
$1002	$06	
$1003	$40	
$1004	$40	
$1005	$4F	
$1006	$55	
$1007	$5A	
$1008	$A2	
$1009	$CE	
$100A	$FA	

CHAPTER 11. CODE CONVERSION FROM INPUT

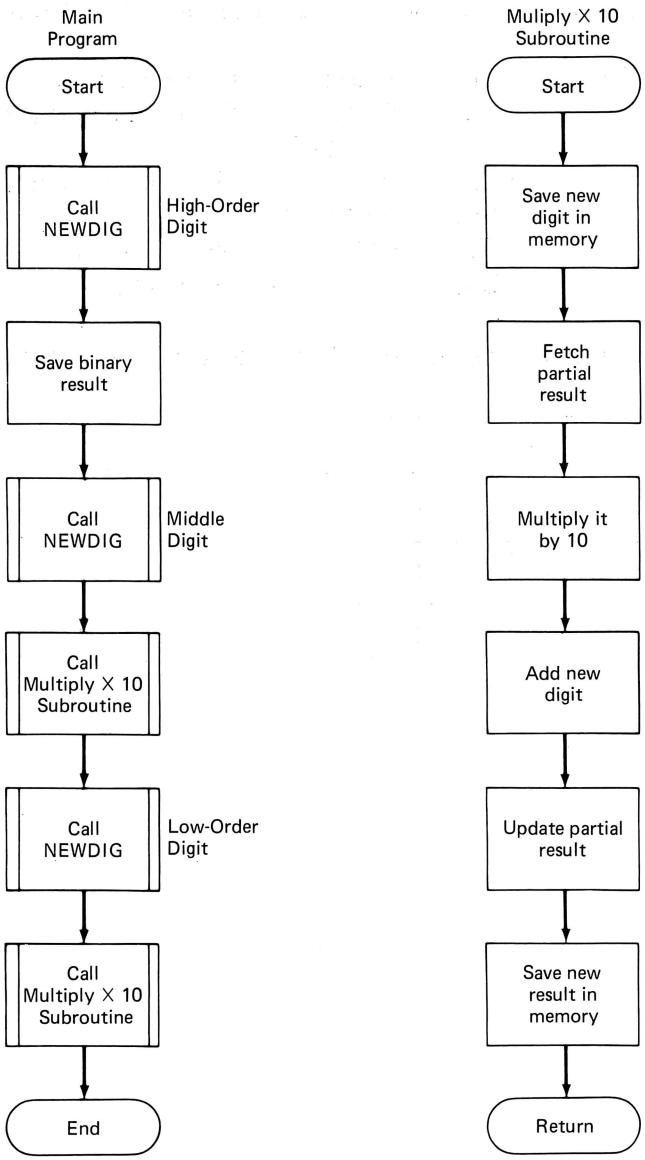
1. A program that processes a single numeric digit from the keyboard is:

```
NEWDIG   JSR   $FD1B     ;Call KEYIN to read digit
         CMP   #$B0      ;Character less than $B0
         BCC   NEWDIG
         CMP   #$BA      ;No.  Is is greater than $B9?
         BCS   NEWDIG
         AND   #$0F      ;No.  Strip off the four MSBs
         BRK             ; and return
```

2. The correct answers for the three execution runs are:

Press Key(s)	Accumulator
9	$09
N, then 4	$04
2	$02

3. a. The flowchart for Step 3 is:



b. The program is:

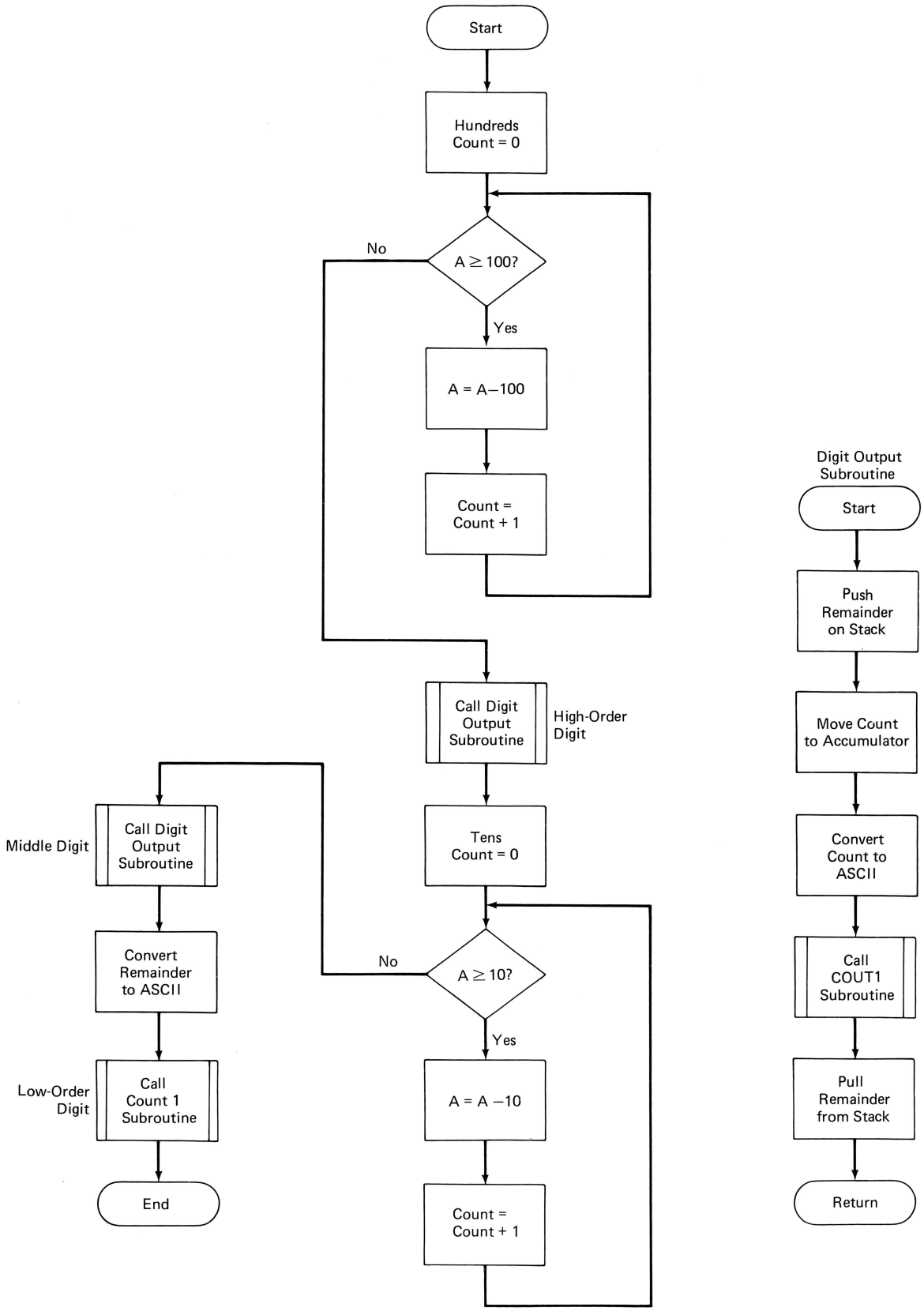
```
            JSR   NEWDIG   ;Fetch high-order digit
            STA   $10      ; and store it as a partial re-
                           sult
            JSR   NEWDIG   ;Fetch middle digit
            JSR   MULT10   ; and add it to partial result
            JSR   NEWDIG   ;Fetch low-order digit
            JSR   MULT10   ; and form final result
            BRK
            ..
            ..
NEWDIG      (Step 1), in subroutine form, goes here
            ..
            ..
*The subroutine to follow multiplies the partial result
*by 10, then adds the new digit to the low-order digit
*position.

MULT10      STA   $11      ;Save digit
            LDA   $10      ;Fetch partial result
            ASL   A        ;Multiply it by two
            ASL   A        ;Multiply it by two again
                           (Total = x 4)
            ADC   $10      ;Add original result to it (Tot-
                           al = x 5)
            ASL   A        ;Multiply this by two (Total
                           = x 10)
            ADC   $11      ;Add new digit
            STA   $10      ; and update partial result
            RTS
```

4. The answers to the four test cases are:

Entered Number	Result (Hex.)	Carry (Yes/No)
000	00	No
120	78	No
255	FF	No
256	00	Yes

CHAPTER 12. CODE CONVERSION FOR OUTPUT

1. a. The flowchart should look like this:



b. This program converts the binary value in the accumulator to a three-digit ASCII decimal string:

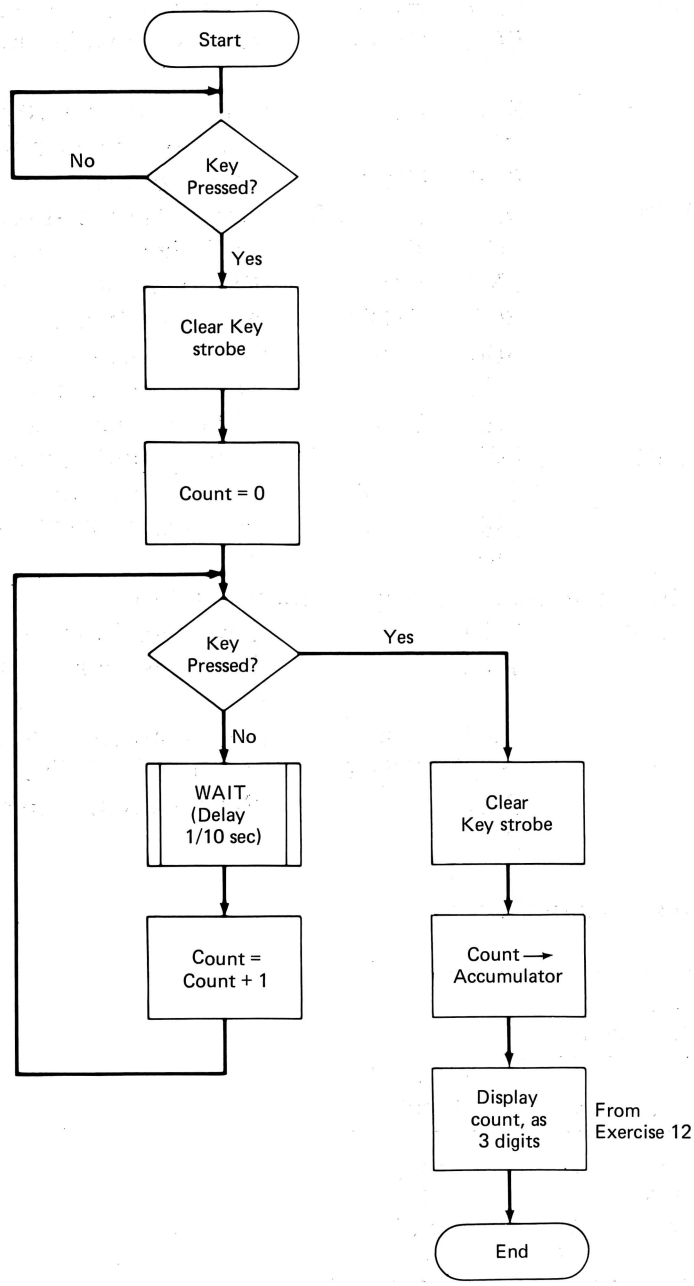
```
        LDX   #0      ;Initialize hundreds counter
C100    CMP   #100    ;Binary value = or greater than 100?
        BCC   OUT1    ;No.  Go display hundreds digit =
                      0
        SBC   #100    ;Yes.  Subtract 100
        INX           ; and increment hundreds count,
        JMP   C100    ; then compare again
OUT1    JSR   PUTOUT  ;Convert and display hundreds digit
        LDX   #0      ;Initialize tens counter
C10     CMP   #10     ;Binary value = or greater than 10?
        BCC   OUT2    ;No.  Go display tens digit = 0
        SBC   #10     ;Yes.  Subtract 10
        INX           ; and increment tens count,
        JMP   C10     ; then compare again
OUT2    JSR   PUTOUT  ;Convert and display tens digit
        ORA   #$B0    ;Convert remainder to ASCII
        JSR   $FDF0   ; and call COUT1 to display it
        BRK
*The convert-and-display subroutine follows:
PUTOUT  PHA           ;Save remainder on stack
        TXA           ;Move decimal count to Accumu-
                      lator
        ORA   #B0     ; and convert it to ASCII,
        JSR   $FDF0   ; then call COUT1 to display it
        PLA           ;Retrieve remainder from stack
        RTS
```

2. The results should be:

Number in Accumulator		Displayed Result
Dec.	Hex.	
030	1E	030
255	FF	255
002	02	002
126	7E	126

CHAPTER 13. TIMING EVENTS WITH SOFTWARE

1. a. The flowchart for the keystroke-timing program is:



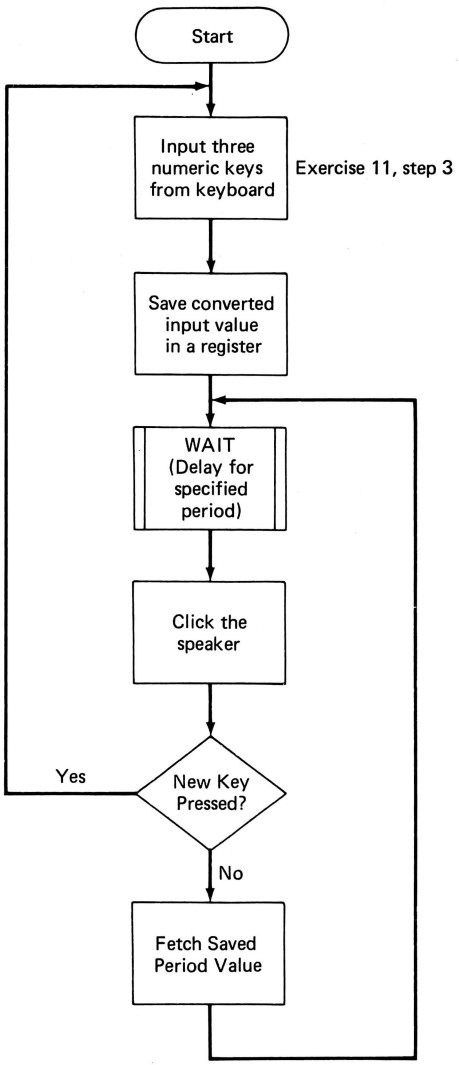
b. The keystroke-timing program is:

```
TIMEKY    BIT   $C000     ;Is a key being pressed?
          BPL   TIMEKY    ;No.  Wait for one
          BIT   $C010     ;Yes. Clear key strobe
          LDY   #0        ;Count = 0
KEY2      BIT   $C000     ;Is a second key pressed
          BMI   SHOCNT    ;Yes.  Go display time count
          LDA   #$C8      ;No.  Delay 1/10 second
          JSR   $FCA8     ; with WAIT subroutine
          INY             ;Update counter
          JMP   KEY2      ;Resume waiting for key
SHOCNT    BIT   $C010     ;Clear key strobe
          TYA             ;Transfer count into Accumulator
          ..
          ..              (Remainder of the program is as
                          given
          ..               in Step 1.b of Exercise 12)
```

3. Normally, pressing two keys should take 0.3 second and pressing the same key twice should take 0.2 second.

CHAPTER 14. APPLE II SOUND

1. a. The flowchart for a program that accepts the period value from the keyboard is shown below.

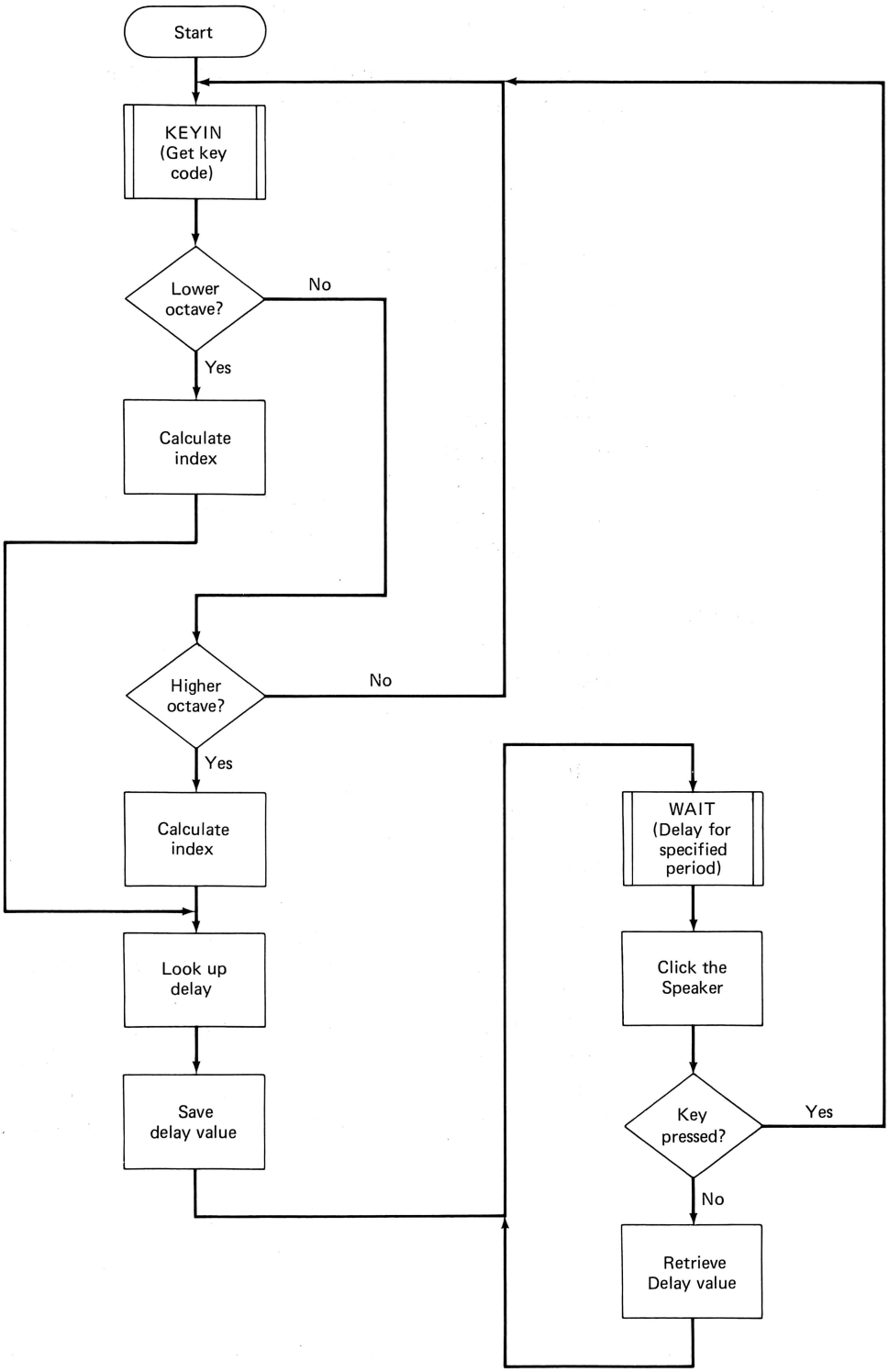


b. The program is as follows:

```
*This program accepts a three-digit decimal number from
*the keyboard, converts this number to binary, and uses
*the result as a time delay between clicks of the
*speaker.
*The program is an endless loop exit by pressing RESET.
*The program affects the A and X Registers.

SOUND    JSR   NEWDIG
         STA   $10
         JSR   NEWDIG
         JSR   MULT10
         JSR   NEWDIG
         JSR   MULT10
         JMP   SAVEA
NEWDIG   JSR   $FD1B
         CMP   #$B0
         BCC   NEWDIG
         CMP   #$BA
         BCS   NEWDIG      } This is taken from Step 3
         AND   #$0F          of exercise 11
         RTS
MULT10   STA   $11
         LDA   $10
         ASL   A
         ASL   A
         ADC   $10
         ASL   A
         ADC   $11
         STA   $10
         RTS

SAVEA    TAX               ;Save period in X
DELAY    JSR   $FCA8       ;WAIT for specified period
         STA   $C030       ;Click the speaker
         BIT   $C000       ;Is a key being pressed?
         BMI   SOUND       ;Yes.  Go read a new period
         TXA               ;No.  Reload Accumulator
         JMP   DELAY       ; and resume with current period
```

CHAPTER 15. LET THERE BE MUSIC!

1. a. The flowchart for the natural notes program should look like this:



b. The source code for the program is:

```
MUSIC   JSR   $FD1B          ;Get a key, with KEYIN
        CMP   #$C1            ;Lower octave?
        BCC   HIOCT           ;No.  Check for upper octave
        CMP   #$C8
        BCS   MUSIC
        AND   #7              ;Yes.  Calculate index
        JMP   INDEX
HIOCT   CMP   #$81            ;Upper octave?
        BCC   MUSIC           ;No.  Get new key
        CMP   #$88
        BCS   MUSIC
        AND   #7              ;Yes.  Calculate index
        ADC   #7
INDEX   TAX                   ;Transfer index to X
        LDA   DLYTBL-1,X      ;Look up delay value
        TAY                   ;and save it in Y
PERIOD  JSR   $FCA8           ;WAIT for specified period
        STA   $C030           ;Click the speaker
        BIT   $C000           ;Is another key being pressed?
        BMI   MUSIC           ;Yes.  Go read it in
        TYA                   ;No.  Continue
        JMP   PERIOD          ; with same delay value
DLYTBL  DFB   $28,$26,$35,$32,$2F,$2E,$2B  ;Lower oct., A-G
        DFB   $1C,$1A,$25,$23,$21,$20,$1E  ;Upper oct., A-G
```

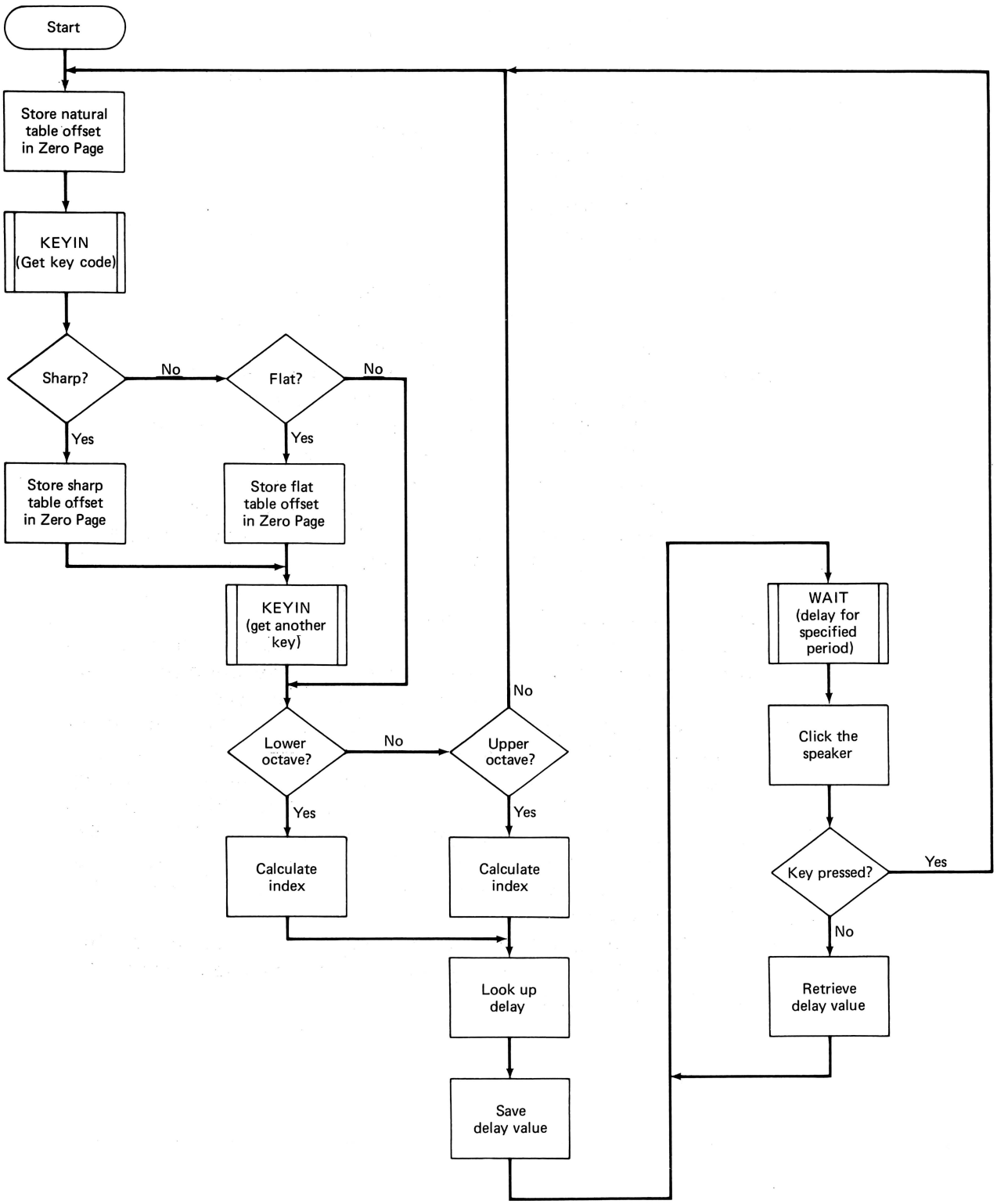
2. The tune is "Mary Had a Little Lamb."
3. One reason why the tones are off-key is the added delay induced by the four instructions at the end of the program. Their execution times were not included in the delay calculations that produced Table 15–1.

   More important, though, the tones are off-key because the accumulator can only hold 256 different delay values (0 through $FF). With only 256 values to choose from, the WAIT subroutine cannot generate a precisely timed delay for each and every note. For example, low C (the first note in Table 15–1) should have a frequency of 130.8 cycles/second, but the $35 value in A generates a frequency of 132.0 cycles/second. Similarly, middle C should be 261.7 cycles/second, but a frequency of 260.0 cycles/second is actually generated.

   One possible solution is to develop a different, more precise substitute for WAIT, using a *2-byte* input value. (You know what comes next: "This exercise is left to the reader.")

CHAPTER 16. MORE MUSIC

1. a. The flowchart for the full music program is:



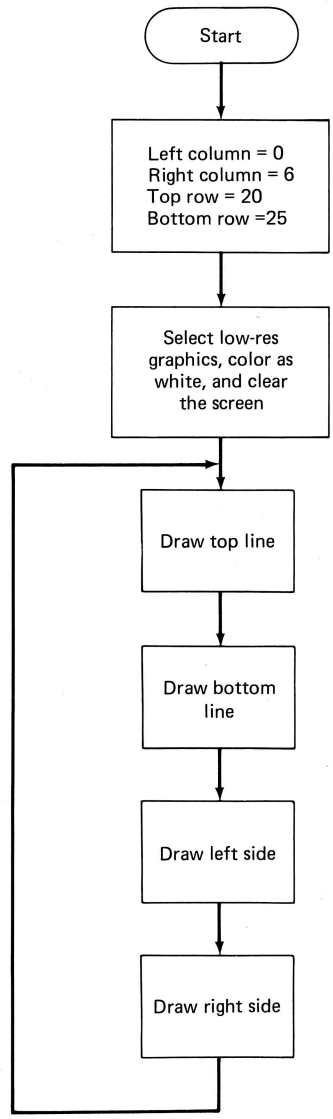
b. The source code for the full music program is:

```
*This program accepts two octaves of notes from
*the keyboard and sounds them on the Apple's speaker.
*The upper span is selected with the CTRL key. Sharps
*are selected by pressing the 3 key, then the letter
*key. Flats are selected by pressing the 4 key, then
*the letter key.
*This program affects the A, X and Y Registers, and
*location $10.

MUSIC1    LDX  #0              ;Store natural table offset
          STX  $10             ; in Page Zero
          JSR  $FD1B           ;Get new key, with KEYIN
          CMP  #$B3            ;Is it the Sharp key (3)?
          BNE  CHKFLAT
          LDX  #DSHARP-DLYTBL  ;Yes.  Store Sharp table
          STX  $10             ; offset in Page Zero
          JMP  MUSIC2
CHKFLAT   CMP  #$B4            ;Is it the Flat key (4)?
          BNE  LOOCT           ;No.  See what key it is
          LDX  #DFLAT-DLYTBL   ;Yes.  Store Flat table
          STX  $10             ; offset in Page Zero
MUSIC2    JSR  $FD1B           ;Get another key, with
                               KEYIN
LOOCT     CMP  #$C1            ;Lower octave?
          BCC  HIOCT           ;No.  Check upper octave
          CMP  #$C8
          BCS  MUSIC1
          AND  #7              ;Yes.  Index = 3 LSBs
          ADC  $10             ; + table offset
          JMP  INDEX
HIOCT     CMP  #$81            ;Upper octave?
          BCC  MUSIC1          ;No.  Invalid key, start over
          CMP  #$88
          BCS  MUSIC1
          AND  #7              ;Yes.  Index = 3 LSBs
          ADC  #7              ; + 7
          ADC  $10             ; + table offset
INDEX     TAX                  ;Transfer index to X
          LDA  DLYTBL-1,X      ;Look up delay value
          TAY                  ; and save it in Y
```

```
PERIOD    JSR   $FCA8          ;WAIT for specified period
          STA   $C030          ;Click the speaker
          BIT   $C000          ;Is another key being
                               pressed?
          BMI   MUSIC1         ;Yes.  Go read it in
          TYA                  ; No.  Continue
          JMP   PERIOD
DLYTBL    DFB   $28,$26,$35,$32,$2F,$2E,$2B
          DFB   $1C,$1A,$25,$23,$21,$20,$1E
DSHARP    DFB   $27,$25,$34,$31,$2E,$2C,$2A
          DFB   $1B,$19,$24,$22,$20,$1F,$1D
DFLAT     DFB   $2A,$27,$36,$34,$31,$2F,$2C
          DFB   $1D,$1B,$26,$24,$22,$21,$1F
```

CHAPTER 17. LOW-RESOLUTION GRAPHICS

1. a. The flowchart should look like this (the order in which the rectangle's sides are drawn is, of course, arbitrary):



b. The rectangle-drawing program should look like this:

```
HLINE     EQU  $F819
VLINE     EQU  $F828
CLRSCR    EQU  $F832
SETCOL    EQU  $F864
*
*Initialize low-resolution graphics
*
          ORG  $300
          LDA  $C052      ;Select full-screen display
          LDA  $C054      ; of page 1
          LDA  $C056      ; in low-resolution graphics
          JSR  CLRSCR     ;Clear the screen
          LDA  #15        ;Set color to white
          JSR  SETCOL
          LDA  $C050      ;Select Graphics mode
*
*Main program starts here
*
DRAW      LDA  TVERT      ;Draw top of rectangle
          LDY  LHOR
          LDX  RHOR
          STX  $2C
          JSR  HLINE
          LDA  BVERT      ;Draw bottom of rectangle
          LDY  LHOR
          JSR  HLINE
          LDY  LHOR       ;Draw left side of rectangle
          LDA  TVERT
          LDX  BVERT
          STX  $2D
          JSR  VLINE
          LDY  RHOR       ;Draw right side of rectangle
          LDA  TVERT
          JSR  VLINE
          JMP  DRAW       ;Wait for RESET
*
*Row and column coordinates
*
LHOR      DFB  0          ;Left column = 0
RHOR      DFB  6          ;Right column = 6
TVERT     DFB  20         ;Top row = 20
BVERT     DFB  25         ;Bottom row = 25
```

2. The source code for this program is nothing more than the source code you developed in step 1b combined with that shown in Figure 17–4. It should look like this:

```
HLINE     EQU  $F819
VLINE     EQU  $F828
CLRSCR    EQU  $F832
SETCOL    EQU  $F864
WAIT      EQU  $FCA8
*
*Initialize low-resolution graphics
*
          ORG  $300
          LDA  $C052     ;Select full-screen display
          LDA  $C054     ; of page 1
          LDA  $C056     ; in low-resolution graphics
          JSR  CLRSCR    ;Clear the screen
          LDA  $C050     ;Set Graphics mode
*
*Main program starts here
*
START     LDA  #15       ;Set color to white
          JSR  SETCOL
          JSR  DRAW      ; and draw the rectangle
          LDX  #3        ;Display it for 0.3 seconds
DELAY     LDA  #200
          JSR  WAIT
          DEX
          BNE  DELAY
          LDA  #0        ;Set color to black
          JSR  SETCOL
          JSR  DRAW      ; and erase the rectangle
          CPY  #39       ;End of screen
          BEQ  BREAK     ;If so, exit
          INC  LHOR      ;If not, increment left and
          INC  RHOR      ; right columns
          JMP  START     ; and start over
BREAK     LDA  $C051     ;Select Text mode
          BRK            ;and return to Monitor
          ..
          ..   (The remainder is the DRAW sequence
          ..    from Step 1.b. in subroutine form
                and the row and column coordinates)
```

CHAPTER 18. RESET AND INTERRUPTS

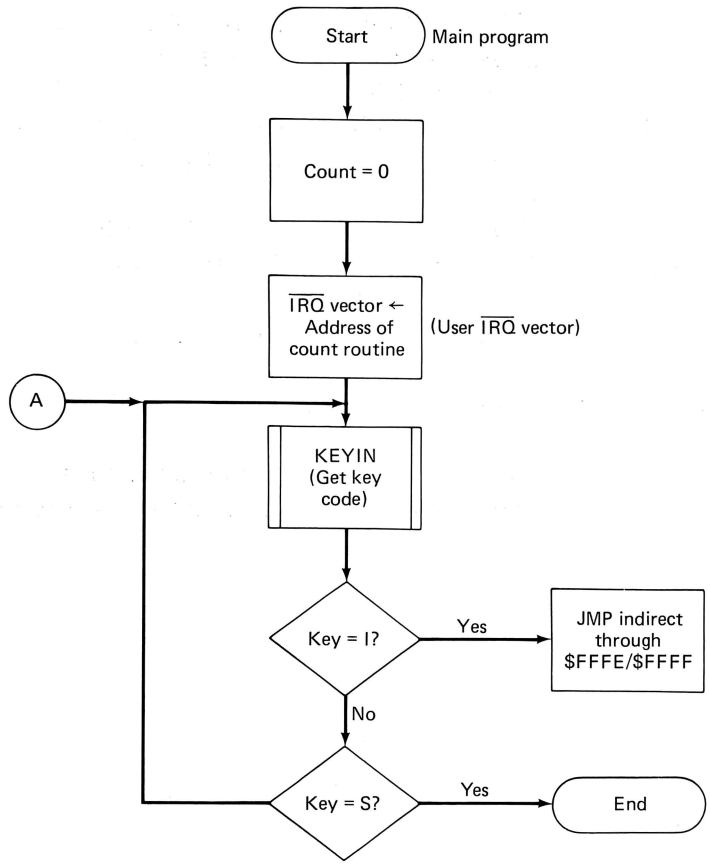
1. The register values should be unchanged.
2. The Apple II Plus used for this book produces the following initial values in the registers:

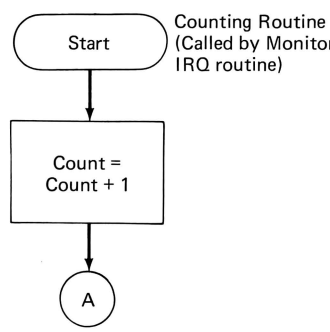
   A = 98 X = 84 Y = D8 P = 00 S = B7

   The exact values in these registers are unimportant. What *is* important is that the registers are *not* initialized to zero or to any other predictable value. You must assume that they are totally random and must be set to some specific value before being used.
3. To force the RESET key to call up the Monitor, you must store the address $FF69 in the soft entry vector (locations $EF2 and $EF3) and store the exclusive-OR of $FF with $A5 in the power-up byte (location $3F4). Therefore, the correct values in memory are as follows:

   ($3F2) = $69
   ($3F3) = $FF
   ($3F4) = $5A

4. Here are the flowcharts for the main program and the counting routine in step 4.





The program that interprets the I key as an interrupt looks like this:

```
            LDX   #00       Count = 0
            STX   $03FE     Store address $1000 into
            LDA   #$10       user IRQ vector
            STA   $03FF
GETKEY      JSR   $FD1B     Get a key, with KEYIN
            CMP   #$C9      Is it the I key?
            BNE   CHKS
            JMP   ($FFFE)   Yes.  Jump indirect to IRQ routine
CHKS        CMP   #$D3      Is it the S key?
            BNE   GETKEY
            BRK             Yes.  Break
```

And the count routine that's called by I is, simply:

```
            INX             Increment count
            JMP   GETKEY     and return for the next key
```

APPENDIX A

Hexadecimal and Decimal Conversion

HEXADECIMAL COLUMNS											
6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
7654		3210		7654		3210		7654		3210	
Byte				Byte				Byte			

POWERS OF 2

2^n	n
256	8
512	9
1 024	10
2 048	11
4 096	12
8 192	13
16 384	14
32 768	15
65 536	16
131 072	17
262 144	18
524 288	19
1 048 576	20
2 097 152	21
4 194 304	22
8 388 608	23
16 777 216	24

$2^0 = 16^0$
$2^4 = 16^1$
$2^8 = 16^2$
$2^{12} = 16^3$
$2^{16} = 16^4$
$2^{20} = 16^5$
$2^{24} = 16^6$
$2^{28} = 16^7$
$2^{32} = 16^8$
$2^{36} = 16^9$
$2^{40} = 16^{10}$
$2^{44} = 16^{11}$
$2^{48} = 16^{12}$
$2^{52} = 16^{13}$
$2^{56} = 16^{14}$
$2^{60} = 16^{15}$

POWERS OF 16

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10
17 592 186 044 416	11
281 474 976 710 656	12
4 503 599 627 370 496	13
72 057 594 037 927 936	14
1 152 921 504 606 846 976	15

APPENDIX B

Computer Numbering Systems

Having spent your entire postadolescent life counting with decimal numbers (0, 1, 2, 3, and so on), it tends to come as a mild shock when you delve into programming books and encounter references to "binary numbers" and "hexadecimal numbers." Have these terms been concocted by computer scientists to keep you in the dark? No, they refer to numbering systems that are much more appropriate than decimal when dealing with computers. Therefore, they are terms that you should understand if you intend to use your computer to its fullest potential.

This appendix is a crash course in computer numbering systems. After learning these concepts, you will have completed a very important first step toward becoming "computer literate."

THE BINARY NUMBERING SYSTEM

In a computer, all information used in programs (instructions and data) is stored in the computer's *memory*. Memory comprises a large number of electrical components that act as light switches, that is, these components have only two possible settings, on or off. However, with only these two settings, combinations of memory components can very effectively represent numbers of any magnitude. How? Read on to find out.

The on and off settings of the components in memory correspond to the two digits of the *binary numbering system*, the fundamental system for computers. Having only two digits, 1 (on) and 0 (off), the binary numbering system is a *base 2* system. This contrasts with the familiar decimal numbering system, which has ten digits (0 through 9), making it a base 10 system.

The light switch-like memory components mentioned earlier are called "bits," which is short for *bi*nary digi*ts*. By convention, a bit that is on represents the value 1 and a bit that is off represents the value 0. This appears to be woefully limiting, until you consider that a decimal digit (no, it's not called a "dit") can only range from 0 to 9. Just as decimal digits can be combined to form numbers larger than 9, binary digits can be combined to form numbers larger than 1.

That is, to represent a decimal number larger than 9 requires another digit, a "tens position" digit. Likewise, to represent a decimal number larger than 99 requires a

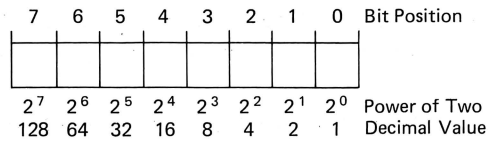


FIGURE B–1. Weights of Eight Binary Digits

"hundreds position" digit, and so on. Therefore, each decimal digit has a *weight* of ten times the digit to its immediate right.

For example, the decimal number 324 can be represented as

$$(3 \times 100) + (2 \times 10) + (4 \times 1)$$

or

$$324 = (3 \times 10^2) + (2 \times 10^1) + (4 \times 10^0)$$

So each decimal digit is a power of ten greater than the preceding digit.

A similar rule applies to the binary numbering system, that is, *each binary digit is a power of two greater than the preceding binary digit.* The rightmost digit has a weight of 2^0, the next digit has a weight of 2^1, and so on. Therefore, the binary number 101 has a decimal value of 5 because

101
$= 1 \times 2^0 = 1$
$= 0 \times 2^1 = 0$
$= 1 \times 2^2 = 4$
Total = 5

Do you now understand how binary numbers are constructed? For each binary digit position, you double the weight of the preceding digit. Thus the first eight binary weights are 1, 2, 4, 8, 16, 32, 64, and 128. These weights are summarized in Figure B–1.

To test your skill at binary numbering, try to construct the binary representations of the decimal values 12, 17, 45, and 79. (You should obtain the binary values 1100, 10001, 101101, and 1001111, respectively.) Conversely, what decimal values are represented by the binary values 1000, 10101, and 11111? (The answers are 8, 21, and 31.)

Eight Bits Form a Byte

The Apple II, the Commodore PET/CBM, the Radio Shack TRS-80, and virtually every other popular microcomputer is designed around an 8-bit *microprocessor.* These microprocessors process 8 bits of information in a single operation. To process more than 8 bits requires additional operations.

In computer literature, an 8-bit quantity of information is called a *byte*. With 8 bits, a byte can represent decimal values from 0 (binary 00000000) to 255 (binary 11111111).

Because a byte is the fundamental unit of processing, microcomputers are described in terms of the number of bytes (rather than bits) contained in their memories. Further, microcromputer manufacturers generally construct memory in blocks of 1,024 bytes. This particular quantity is an outgrowth of the binary orientation of computers, in that it represents exactly 2^{10} bytes.

The value 1,024 has also been given an industry standard abbreviation; it is referred to by the letter K. Thus, when you read an advertisement for a computer that has a "16K RAM," the manufacturer is telling you that this particular product has 16 × 1,024 (or 16,384) bytes of programmable memory.

THE HEXADECIMAL NUMBERING SYSTEM

Although the binary numbering system is an accurate way to represent numbers in memory, strings of nothing but ones and zeroes become very difficult to work with for any period of time. They are very prone to errors, too, because a number such as 10110101 is extremely easy to write incorrectly as 10110110 (the rightmost two bits have been transposed).

Years ago, programmers found that they were always required to operate on *groups* of bits rather than individual bits. The earliest processors were 4-bit devices (they processed information 4 bits at a time), so the logical alternative to binary numbering was a system that numbered bits in groups of four.

As you know, 4 bits can represent the binary values 0000 through 1111, which is equivalent to the decimal values 0 through 15. If each digit of a numbering system is to represent 4 bits, that numbering system will have digits that can range from 0 to 15 (decimal). Therefore, that numbering system is a *base 16* system.

If binary is used to denote a base 2 system and decimal is used to denote a base 10 system, what term can be used to denote a base 16 system? Well, whoever named the base 16 system took the Greek word *hex* (for six) and the Latin word *decem* (for ten), and combined them to form the word *hexadecimal*. Thus, the base 16 numbering system is called the *hexadecimal numbering system*.

The 16 digits of the hexadecimal numbering system are labeled 0 through 9 (decimal values 0 through 9) and A through F (decimal values 10 through 15). The decimal equivalents are summarized in Table B–1 for reference purposes.

Like binary and decimal digits, each hexadecimal digit has a "weight" that is some power of its base. Since 16 is the base for the hexadecimal numbering system, each hexadecimal digit has a weight that is a power of 16 higher than the digit to its immediate right; that is, the rightmost digit has a weight of 16^0, the second digit has a

TABLE B–1. Hexadecimal Numbering System

Hexadecimal Digit	Decimal Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

weight of 16^1, and so on. For example, the hexadecimal value 3AF has a decimal value of 943 because

$$
\begin{array}{l}
3AF \\
\quad = F \times 16^0 = 15 \times 1 = 15 \\
\quad = A \times 16^1 = 10 \times 16 = 160 \\
\quad = 3 \times 16^2 = 3 \times 256 = 768 \\
\qquad\qquad \text{Total} = 943
\end{array}
$$

Use of Hexadecimal Numbers

The primary use of hexadecimal numbers is in referring to memory locations. Most 8-bit microprocessors have an addressing range of 65,536 bytes; that is, 64K bytes. This means that memory locations can have addresses from 0 to 65,535 decimal or from 0 to FFFF hexadecimal.

As a rule, the manufacturer's literature numbers memory locations in hexadecimal form, so in order to be comfortable with that literature you must be conversant with the hexadecimal numbering system. For most of your work it is unnecessary to know how to add, subtract, or otherwise process hexadecimal numbers, but you should know that when the literature refers to memory location FF00, for instance, it is referring to decimal location 65280.

A FINAL COMMENT ABOUT NUMBERING SYSTEMS

It is important to realize that the various numbering systems are simply alternate ways for you to *represent* the information that is being processed by the computer. The fact that you choose to represent that information in binary rather than in decimal, or in hexadecimal rather than in binary, has absolutely no effect on how the computer's *microprocessor* operates on that information. To a microprocessor, the information in memory is nothing more than a series of binary patterns which are processed according to the machine code instruction that is currently being executed.

APPENDIX C

ASCII Character Set for Apple II

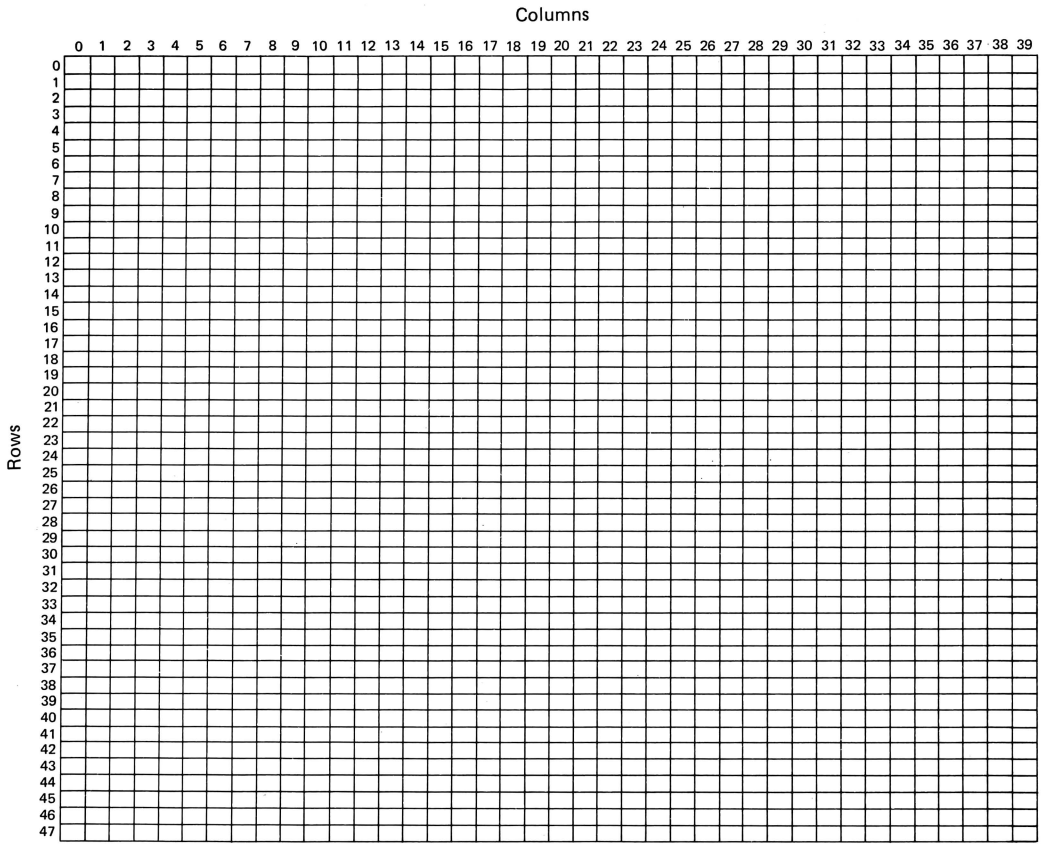
LSD \ MSD		8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	•	>	N	↑	n	~
F	1111	SI	US	/	?	O	←	o	DEL

NUL — Null
SOH — Start of Heading
STX — Start of Text
ETX — End of Text
EOT — End of Transmission
ENQ — Enquiry
ACK — Acknowledge
BEL — Bell
BS — Backspace
HT — Horizontal Tabulation
LF — Line Feed
VT — Vertical Tabulation
FF — Form Feed
CR — Carriage Return
SO — Shift Out
SI — Shift In

DLE — Data Link Escape
DC — Device Control
NAK — Negative Acknowledge
SYN — Synchronous Idle
ETB — End of Transmission Block
CAN — Cancel
EM — End of Medium
SUB — Substitute
ESC — Escape
FS — File Separator
GS — Group Separator
RS — Record Separator
US — Unit Separator
SP — Space (Blank)
DEL — Delete

APPENDIX D

Low-Resolution Graphics Screen



APPENDIX E

6502 Instruction Set Table

INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM			IMPLIED			(IND, X)			(IND), Y			Z. PAGE, X			ABS. X			ABS. Y			RELATIVE			INDIRECT			Z. PAGE, Y			PROCESSOR STATUS CODES	
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7 6 5 4 3 2 1 0 N V • B D I Z C	MNEMONIC
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										N V • • • • Z C	ADC
AND	A∧M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										N • • • • • Z •	AND
ASL	C ← 7 0 ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3												N • • • • • Z C	ASL
BCC	BRANCH ON C = 0 (2)																															90	2	2							• • • • • • • •	BCC
BCS	BRANCH ON C = 1 (2)																															B0	2	2							• • • • • • • •	BCS
BEQ	BRANCH ON Z = 1 (2)																															F0	2	2							• • • • • • • •	BEQ
BIT	A∧M				2C	4	3	24	3	2																															M_7 M_6 • • • Z •	BIT
BMI	BRANCH ON N = 1 (2)																															30	2	2							• • • • • • • •	BMI
BNE	BRANCH ON Z = 0 (2)																															D0	2	2							• • • • • • • •	BNE
BPL	BRANCH ON N = 0 (2)																															10	2	2							• • • • • • • •	BPL
BRK	BREAK													00	7	1																								• • • 1 • 1 • •	BRK
BVC	BRANCH ON V = 0 (2)																															50	2	2							• • • • • • • •	BVC
BVS	BRANCH ON V = 1 (2)																															70	2	2							• • • • • • • •	BVS
CLC	0 → C													18	2	1																								• • • • • • • 0	CLC
CLD	0 → D													D8	2	1																								• • • • 0 • •	CLD
CLI	0 → I													58	2	1																								• • • • • 0 • •	CLI
CLV	0 → V													B8	2	1																								• 0 • • • • •	CLV
CMP	A − M	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										N • • • • • Z C	CMP
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																															N • • • • • Z C	CPX
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																															N • • • • • Z C	CPY
DEC	M − 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3												N • • • • • Z •	DEC
DEX	X − 1 → X													CA	2	1																								N • • • • • Z •	DEX
DEY	Y − 1 → Y													88	2	1																								N • • • • • Z •	DEY
EOR	A ∀ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										N • • • • • Z •	EOR
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3												N • • • • • Z •	INC
INX	X + 1 → X													E8	2	1																								N • • • • • Z •	INX
INY	Y + 1 → Y													C8	2	1																								N • • • • • Z •	INY
JMP	JUMP TO NEW LOC				4C	3	3																												6C	5	3				• • • • • • • •	JMP
JSR	JUMP SUB				20	6	3																																		• • • • • • • •	JSR
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										N • • • • • Z •	LDA
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	N • • • • • Z •	LDX
LDY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3												N • • • • • Z •	LDY
LSR	0 → 7 0 → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3												0 • • • • • Z C	LSR
NOP	NO OPERATION													EA	2	1																								• • • • • • • •	NOP
ORA	A ∨ M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										N • • • • • Z •	ORA
PHA	A → Ms S − 1 → S													48	3	1																								• • • • • • • •	PHA
PHP	P → Ms S − 1 → S													08	3	1																								• • • • • • • •	PHP
PLA	S + 1 → S Ms → A													68	4	1																								N • • • • • Z •	PLA
PLP	S + 1 → S Ms → P													28	4	1																								(RESTORED)	PLP
ROL	7 0 C				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3												N • • • • • Z C	ROL
ROR	C 7 0				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3												N • • • • • Z C	ROR
RTI	RTRN INT													40	6	1																								(RESTORED)	RTI
RTS	RTRN SUB													60	6	1																								• • • • • • • •	RTS
SBC	A − M − $\overline{C}$ → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										N V • • • • Z (3)	SBC
SEC	1 → C													38	2	1																								• • • • • • • 1	SEC
SED	1 → D													F8	2	1																								• • • • 1 • • •	SED
SEI	1 → I													78	2	1																								• • • • • 1 • •	SEI
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										• • • • • • • •	STA
STX	X → M				8E	4	3	86	3	2																														96	4	2	• • • • • • • •	STX
STY	Y → M				8C	4	3	84	3	2													94	4	2																• • • • • • • •	STY
TAX	A → X													AA	2	1																								N • • • • • Z •	TAX
TAY	A → Y													A8	2	1																								N • • • • • Z •	TAY
TSX	S → X													BA	2	1																								N • • • • • Z •	TSX
TXA	X → A													8A	2	1																								N • • • • • Z •	TXA
TXS	X → S													9A	2	1																								• • • • • • • •	TXS
TYA	Y → A													98	2	1																								N • • • • • Z •	TYA

(1) ADD 1 to "N" IF PAGE BOUNDARY IS CROSSED

(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE

(3) CARRY NOT = BORROW

(4) IF IN DECIMAL MODE, Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X	INDEX X	+	ADD	M_7	MEMORY BIT 7
Y	INDEX Y	−	SUBTRACT	M_6	MEMORY BIT 6
A	ACCUMULATOR	∧	AND	n	NO. CYCLES
M	MEMORY PER EFFECTIVE ADDRESS	∨	OR	#	NO. BYTES
Ms	MEMORY PER STACK POINTER	∀	EXCLUSIVE OR		

Computers $9.95

APPLE II®
Assembly Language Exercises

While other books teach you how to program in assembly language, here is the only workbook that gives you the true "hands-on" exercises and problems you need to really refine and perfect your programming skills on the Apple II. The ideal companion to any assembly language programming guide based on the 6502 microprocessor chip, APPLE II® ASSEMBLY LANGUAGE EXERCISES helps you unlock the full power of your Apple and gain an appreciation for good programming practices. *The key is 18 easy-to-read, easy-to-use exercises that put the speed and efficiency of assembly language at your fingertips.*

From a discussion of the assembly language characteristics of the Apple II, APPLE II® ASSEMBLY LANGUAGE EXERCISES guides you through the entire instruction set, letting you master progressively more sophisticated and useful programs every step of the way. You'll find exercises in processing lists and tables, input/output techniques, and even how to develop programs that play music through the Apple's built-in speaker—something that is nearly impossible in BASIC, but an easy matter using assembly language.

For each exercise, you'll draw a flowchart . . . write your program in the book . . . enter and execute the program on your Apple . . . and finally run the program for specified test conditions. And with this step-by-step approach, you'll have a complete, documented record if you need to debug any of your programs.

By the time you complete your APPLE II® ASSEMBLY LANGUAGE EXERCISES, you'll have built a library of assembly language programs and subroutines . . . *and* the expertise to use them to design your own programs.

Leo J. Scanlon is a software consultant and author of several other books on computers and microprocessors.

Apple® is a trademark of Apple Computer, Inc. Any questions concerning the material in this book should be directed to the publisher, John Wiley & Sons, and not to Apple Computer, which is not responsible for and was not involved in the preparation of this book.

More than a million people have learned to use, program, and enjoy microcomputers with Wiley paperback guides. Look for them all at your favorite bookshop or computer store!

Photo: Arnold Katz
Design: Elaine Tannenbaum

JOHN WILEY & SONS, Inc.
605 Third Avenue, New York, N.Y. 10158
New York • Chichester • Brisbane • Toronto • Singapore

ISBN 0 471-86598-2

Scanlon

APPLE II® Assembly Language Exercises

Wiley